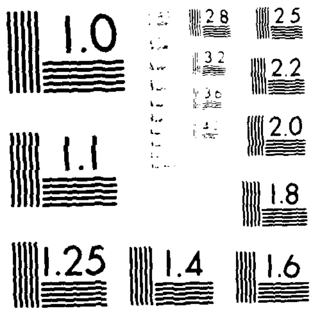

1.0
2.8
2.5
3.2
2.2
3.6
4.0
2.0
1.1
1.8
1.25
1.4
1.6

NSWC TR 84-101

THLL REFERENCE MANUAL

Hartmut G. Huber
Strategic Systems Department

July 1984

NAVAL SURFACE WEAPONS CENTER

Dahlgren, Virginia 22448

NSWC TR 84-101

THLL REFERENCE MANUAL

Hartmut G. Huber
Strategic Systems Department

July 1984

NAVAL SURFACE WEAPONS CENTER

Dahlgren, Virginia 22448

UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER NSWC TR 84-101	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) THLL REFERENCE MANUAL		5. TYPE OF REPORT & PERIOD COVERED Final
		6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) Hartmut G. Huber		8. CONTRACT OR GRANT NUMBER(s)
9. PERFORMING ORGANIZATION NAME AND ADDRESS Naval Surface Weapons Center (Code K53) Dahlgren, VA 22448		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS 36801
11. CONTROLLING OFFICE NAME AND ADDRESS Strategic Systems Program Office Washington, DC 20376		12. REPORT DATE July 1984
		13. NUMBER OF PAGES 210
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office)		15. SECURITY CLASS. (of this report) UNCLASSIFIED
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution is unlimited

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

TRIDENT Higher Level Language	VAX 11/780	program
THLL	procedure	
TRIDENT Digital Control Computer (TDCC)	block-structure	
MC68000	Compiler	

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

The TRIDENT Higher Level Language (THLL) is a procedure-oriented, block-structured programming language used for writing operational software for the TRIDENT Digital Control Computer (TDCC) and the MC68000 as well as support software for the VAX 11/780. This Technical Report serves as a complete reference manual for THLL. Target machine dependencies are, in general, explained in this document for each of the three target machines. The runtime support and target machine unique features are documented in References 1, 2, and 3.

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE
S/N 0102-LF-014-6601

UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

FOREWORD

This document was written in the Operational Support Branch (K53), Submarine Launched Ballistic Missile (SLBM) Software Development Division (K50), of the Strategic Systems Department (K) at the Naval Surface Weapons Center (NSWC), Dahlgren, Virginia.

The purpose of this report is to serve as a complete reference manual to the users of the TRIDENT Higher Level Language (THLL). It describes all elements of THLL the user has to know in order to write THLL programs for any one of the three target machines: the TRIDENT Digital Control Computer (TDCC), the MC68000, or the VAX11/780. Target machine unique information can be found in References 1, 2, and 3.

This report is based on NSWC Technical Report TR-3657, (Revised June 1978). Much of the material in TR-3657 was rewritten and reorganized. This report was extensively reviewed by personnel in the Operational Support Software Systems Group in K53, the Quality Assurance Branch (K52), and the Operational Systems Branch (K54).

This project was funded by the Strategic Systems Program Office, Washington, DC 20376, under task number 36801.

Questions, comments, and suggestions concerning the material presented in this document should be directed to the Commander, Naval Surface Weapons Center, ATTN: K53, Dahlgren, Virginia 22448.

Approved by:

THOMAS A. CLARE, Head
Strategic Systems Department

CONTENTS

TABLES

CHAPTER 1

INTRODUCTION

1.1 INTRODUCTION TO THE LANGUAGE

The TRIDENT Higher Level Language (THLL) is a procedure-oriented, block-structured programming language. This Reference Manual provides a complete description of the syntax and semantics of THLL.

The meaning of most features in THLL is independent of the particular target machine on which a THLL program executes. Some features, however, do depend on the target machine. Those dependencies are explained for each of the following target machines: TRIDENT Basic Processor, TRIDENT MC68000, and VAX.

When programs execute on a particular target machine the runtime environment is determined by the underlying Operating System and a collection of runtime support procedures which, in general, have machine-dependent characteristics. The machine-dependent runtime environment for THLL programs is described for each target machine in a separate document (References 1, 2, and 3).

This introductory section describes, from an intuitive point of view, how a THLL program is formed from certain basic elements. Precise details about these elements can be found in later sections.

A THLL program that is acceptable to the THLL compiler (TRICOMP) for translation into code for a target machine is called a compile unit. Normally, many compile units are combined by a link loader system to form a self-contained program that can be executed on the target machine.

Every THLL compile unit has the following structure:

```
ID1
     BEGIN
     D1 ;
     D2 ;
      .
      .
      .
     Dn ;
     END FINIS
```

The words BEGIN, END, FINIS are language keywords and have a fixed meaning. Each Di, i=1,...,n, is a declaration, for example a data declaration (see Chapter 3) or a procedure declaration (see Chapter 6). The first BEGIN in a compile unit must be preceded by an identifier, ID1, which is used by the compiler as the compile unit name. For programs which run on the BP, this identifier is also used as the compile unit name by the Symbolic Debug System.

Each declaration defines the meaning of an identifier. A procedure declaration defines an identifier to be the name of a procedure. Data declarations define identifiers to be variables of various types or to be names of more complex data structures such as arrays or stacks.

Examples:

A. Each of the following three lines represents a data declaration terminated by a semicolon:

```
INTEGER I,K,INTVAR ;
REAL X,Y ;
REAL ARRAY A(5,10) ;
```

As long as these declarations are in effect (see scope of ide[illegible]iers, Section 7.1.4), I, K, and INTVAR can be used as integer variables; X, Y as real variables; and A(n1, n2) as a subscripted variable. Here n1 and n2 are integer numbers, $0 \le n1 \le 5$, $0 \le n2 \le 10$, or expressions which evaluate at runtime to integers in these ranges.

B. The following seven lines represent a procedure declaration:

```
DEFINE REAL PROCEDURE DUMMY(X,Y,Z) ;
  VALUE X,Y,Z ;
  INTEGER X ;
  REAL Y,Z ;
     BEGIN
     RETURN IF X LES 0 THEN Y ELSE Z ENDIF ;
     END
```

As long as this declaration is in effect, DUMMY is a procedure which can be called by an expression such as:

```
DUMMY(2,3.14,-.5)
```

The numbers 2, 3.14, and -.5 are called the actual parameters. They may, in general, be expressions which evaluate to numbers of the type INTEGER, REAL, and REAL.

The above procedure declaration defines DUMMY to be the name of a procedure of three arguments. X, Y, and Z are called formal parameters. Their type and method of transmission are specified in the "specification part," as shown in lines 2-4. All parameters are to be passed by value; X has type INTEGER; and Y and Z have type REAL. The last three lines form the body of the procedure representing the sequence of calculations necessary to compute the value of DUMMY. In general, the body of a procedure is a sequence of statements and expressions separated by semicolons and enclosed by BEGIN-END brackets. In our example, this sequence consists of one statement only, a return statement of the form:

```
RETURN e
```

where e is the conditional expression

```
IF X LES 0 THEN Y ELSE Z ENDIF
```

The value of e is the value of Y if X is negative, otherwise the value of Z. The effect of the return statement is to define the value of DUMMY to be the value of e and to return control to the point of call.

The concept of a procedure in THLL corresponds to the mathematical concept of a function but is slightly different. The difference is that a procedure may have side effects on the environment. This means that a procedure can change the value of a variable and this change remains in effect after control returns from the procedure. A second difference is that there may not be a value defined for a procedure. A function, of course, always has a value. An unvalued procedure is executed for its effect on the environment. For simplicity, the concepts of procedure and function are used interchangeably.

In any case, the action elements of the compile unit are always contained in the body of a procedure. It is the procedure body that gives rise to executable code. Declarations just inform the compiler about the meaning of identifiers and do not cause code to be generated.

The action elements in a compile unit are statements and expressions. It is important for the reader to grasp an intuitive understanding of the difference between statements and expressions. Statements and expressions are both executed as one or more elementary actions. An expression results in a value, a statement does not.

Examples:

```
1.  X
2.  X * (Y - 5) + 1
3.  X = 2
4.  Y = (X = 1)
5.  RETURN X = 1
6.  GOTO L
7.  BEGIN
    X = A + B ;
    Y = X - 1 ;
    END
```

Examples 1-4 are expressions; examples 5-7 are statements. Examples 3 and 4 show that in THLL an assignment is an expression. The value of an assignment expression is the value of the right side. Thus, the value of the assignment X = 2 is 2. Similarly, the value of Y = (X = 1) is the value of X = 1, which is 1.

Normally, a THLL compile unit contains one or more procedure declarations. One procedure in a compile unit can be designated as the procedure that receives control initially when the program containing the compile unit is executed. Such a procedure must not have any arguments. The designation of the start-up procedure is machine-dependent.

A THLL compile unit may contain no procedure declarations at all, only data declarations. The compiled unit, of course, cannot be executed since it does not contain any executable code. Its purpose is to define data to be used by other compile units.

One of the most important concepts in THLL is "block." A block is a statement of the following form:

```
BEGIN
D1 ;
 .
 .
 .
Dn ;
e1 ;
 .
 .
 .
ek ;
END
```

where D1; ...; Dn is a (possibly empty) sequence of declarations and e1; ...; ek is a non-empty sequence of expressions or statements (see Section 5.4). A block has two major effects. First, any identifier declared after the BEGIN is made local to the block. The declared meaning of the identifier is valid within the block and all inner blocks (if not redefined), but is not valid outside of it. This range of validity of a declaration is called the scope of a declaration or the scope of the (declared) identifier. The second effect of a block is to make the sequence of computations represented by e1; ...; ek one unit, called a compound statement. Since the elements of a block are again statements, it follows that blocks and statements are recursive in nature and have a nested structure.

Example:

```
BEGIN
INTEGER X,Y ;
X = 1 ;
IF Y NEQ 0 THEN
     BEGIN
     REAL Z ;
      .
      .
      .
     END
ELSE
     BEGIN
      .
      .
      .
     END
ENDIF ;
Y = 2 ;
END
```

1.2 DESCRIPTION OF NOTATION

The following sections present the details of all language constructs in a systematic fashion. Syntactic units are written as a name or as a phrase enclosed in angle brackets < >. Examples are <identifier> or <simple expression>. In defining language constructs for which there are alternative choices, these choices appear in a list enclosed by curly braces { }. In such cases, one of the alternatives must be selected. If an item is optional, it is enclosed by square brackets [ ]. If the optional item has alternatives, the alternatives appear in a list within the brackets.

1.3 COMPATIBILITY OF THLL AND THLL II

THLL is used for TRIDENT I program development. For TRIDENT II an enhanced version of THLL, called THLL II, will be used. Most THLL II enhancements are extensions of THLL, making a THLL program a special case of a THLL II program. However, the introduction of the delimiters ELSEIF, NEXTCASE, and ENDDO into THLL II is not an extension.

To make THLL programs compatible with THLL II programs, this Reference Manual describes the delimiters ELSEIF, NEXTCASE, and ENDDO as they are used in THLL II. Each compile unit that uses these delimiters must contain the following synonym declarations that map these delimiters to the corresponding delimiters for THLL:

```
SYNONYM ELSEIF   = /,/ ;
SYNONYM NEXTCASE = /,/ ;
SYNONYM ENDDO    = / / ;
```

CHAPTER 2

BASIC ELEMENTS OF THLL

2.1 OVERVIEW

On the lowest level, a THLL compile unit is a character string. Characters are grouped together as items. Each item falls into one of the following four categories:

A. Operators,

B. Delimiters,

C. Constants, and

D. Symbols (identifiers).

There is a fixed number of operators and delimiters. They are listed in Tables 2-1 and 2-2. Sections 2.6 and 2.7 describe, respectively, which character sequences are symbols and which are constants of various types.

Examples:

operators	+ =
delimiters	IF ; PROCEDURE
constants	3 TRUE #* STRING CONSTANT *
symbols	TRIDENT SUBR X

The above classifications are based on the meaning of items and on how meanings are attached to the items.

Operators, delimiters, and constants each have a fixed meaning in the language which is known by the compiler. The user has no mechanism for changing this meaning. It can only be changed by modifying the compiler. Operators and delimiters are also called reserved words of THLL.

Symbols, on the other hand, do not, in general, have any meaning by themselves. They must be given a meaning by the user. Declarations provide the mechanism for doing this.

Some symbols have a predefined meaning known by the compiler; for example, the names of standard functions such as SIN, COS, SWA, and SET.BIT or the predefined components WHOLEW, DOUBLEW, and REALDW. These identifiers can be used without defining them in a declaration.

Every symbol can be redefined by writing down a declaration at the beginning of a block. The new meaning is in effect within the scope of the declaration. Of course, the programmer should not specify two different declarations for the same identifier within the same block.

A symbol which is used as a label in a compile unit is considered to be defined as soon as it appears in a label position; that is, the symbol precedes a colon (Section 5.1). For example:

```
DONE:
```

Apart from labels and predefined symbols, every identifier must be declared somewhere by a declaration. Any references to an identifier must lie within the scope of a valid declaration for the identifier. Otherwise the compiler considers the symbol to be undefined.

2.2 THLL CHARACTER SET

The THLL character set consists of the following ASCII subset:

Uppercase letters A-Z

Numerals 0-9

The following special characters:

Character	Name	Character	Name
	Space	.	Period
!	Exclamation point	/	Slash
"	Quotation mark	:	Colon
#	Number sign	;	Semicolon
$	Dollar sign	<>	Angle brackets
%	Percent sign	=	Equal sign
&	Ampersand	?	Question mark
'	Apostrophe	@	At sign
()	Parentheses	[]	Square brackets
*	Asterisk	\	Backslash
+	Plus sign	^	Circumflex
,	Comma	_	Underline
-	Minus sign		

There is no plus or minus sign (±) defined in ASCII and the circumflex (^) is used to represent it. The TDCC devices KBDSS, CPRINT, and SPRINT display the circumflex as the ± character.

Not included in the THLL character set are the lowercase letters (codes X'60' - X'7E') and the nonprintable characters (codes X'00' - X'1F' and X'7E'). When these characters are used in a THLL source file they are treated as follows:

A. Lowercase letters are treated in a machine-dependent manner:

BP, MC68000: Lowercase letters are converted to uppercase letters (codes X'40' - X'5E') except when they appear in comments or remarks.

VAX: Lowercase letters are converted to uppercase letters (codes X'40' - X'5E') except when they appear in THLL strings or in comments or remarks.

B. Nonprintable characters are converted to question marks (?).

A table of ASCII characters is included in Appendix A.

2.3 LETTERS

Letters have no individual meaning. They are used to form identifiers and strings.

2.4 DIGITS

Four sets of digits are available to the THLL user:

A. The <u>binary</u> digits 0 and 1,

B. The <u>octal</u> digits 0 through 7,

C. The <u>decimal</u> digits 0 through 9,

D. The <u>hexadecimal</u> digits 0-9 and A-F.

Digits are used in forming numbers, identifiers, and strings.

2.5 RESERVED WORDS

2.5.1 Operators

An operator denotes a function for which an infix notation is used instead of a standard functional notation. For example:

operator	infix notation	functional notation
+	A + B	+ (A, B)

All operators yield values. Table 2-1 contains a complete list of all operators.

The type of LOC X is POINTER, the type of LOCA X is INTEGER. The value of LOC X (LOCA X) is the address (absolute address) of the THLL word or doubleword containing X. The meaning of address and absolute address is machine-dependent:

BP:
An address is a virtual address (see Reference 1). An absolute address is an actual machine address. In general, these two values are not identical.

MC68000:
An address and an absolute address are both an actual machine address. These two values are always identical.

VAX:
An address and an absolute address are both a virtual address. These two values are always identical.

2.5.2 Delimiters

Delimiters can be used to punctuate, to provide descriptive information, and to indicate structure. Delimiters have fixed meanings and uses. Table 2-2 contains a complete list of delimiters.

The space is used as a delimiter between THLL items. It cannot be used within THLL items.

Example:

If it is desired to compare two values X and Y, increase X by 5 if it is not equal to Y, or decrease X by 1 if it is equal, the sequence of characters would be as follows:

```
IFXNEQYTHENX=X+5ELSEX=X-1ENDIF ;
```

However, this string of characters as processed by the compiler does not produce the desired code because there are no delimiters to distinguish reserved words from variables, operators, etc. To produce the correct code, the string could be:

```
IF X NEQ Y THEN X=X+5 ELSE X=X-1 ENDIF ;
```

TABLE 2-1. THLL OPERATORS

Class	Mnemonic	Meaning
Arithmetic	+	addition
Arithmetic	-	subtraction
Arithmetic	*	multiplication
Arithmetic	/	division
Arithmetic	**	exponentiation
Arithmetic	MOD	remainder on division
Relational	LES	less than
Relational	LEQ	less than or equal
Relational	EQL	equal
Relational	GRT	greater than
Relational	GEQ	greater than or equal
Relational	NEQ	not equal
Logical	OR	or
Logical	XOR	exclusive or
Logical	AND	and
Logical	NOT	not
Assignment	=	assignment of value
Bit	ANDB, BITAND	and bits
Bit	ORB, BITOR	or bits
Bit	XORB, BITXOR	exclusive or of bits
Bit	NOTB, BITNOT	not bits
Addressing	LOC	address of variable
Addressing	LOCA	absolute address
Addressing	ENTRYP	ENTRYP X allows the procedure X to be passed as a parameter.

TABLE 2-2. DELIMITERS

ALPHA	EXTERNAL	OWN
ARITHMETIC	FIELD	POINTER
ARRAY	FINIS	PRESET
BEGIN	FOR	PROCEDURE
CASE	FORMAT	REAL
CASEEND	GLOBAL	REPEAT
COMEND	GOTO	RETURN
COMMENT	HALF	SPRINT
COMMON	ICL	STACK
COMPONENT	IF	STEP
CPRINT	IFEND	SWITCH
DEFINE	INSERT	SYNONYM
DEVICE	INTEGER	TASS
DO	INTERRUPT	THEN
DOUBLE	KBDSS	TO
ELSE	LINK	UNTIL
ELSEIF	LOGICAL	VALUE
END	LOOPEXIT	WHILE
ENDCASE	MDF	;
ENDCOM	MTF	,
ENDDO	NEXTCASE	:
ENDIF	NULL	(
EXEC	OFFSET	)
EXIT	OPTARG	$

2.6 IDENTIFIERS

Identifiers may be used as labels, variables, procedures, switches, formats, stacks, arrays, components, devices, or commons.

An identifier is a sequence of letters, digits, and dots. The first character must be a letter. The last character must not be a dot. The maximum length of an identifier is 256 characters. A compile unit name is an identifier that does not contain special characters.

Different target machines may allow additional special characters. Also, the number of significant characters is machine-dependent:

BP:

The first eight characters are significant. A dot in an external name is mapped into a dollar sign.

VAX:

The special characters dot, underscore, and dollar sign are allowed in identifiers except in the first character position. The first 31 characters are significant. Special characters in external symbols are not mapped to other characters.

MC68000:

The special characters dot, underscore, and question mark are allowed in identifiers except in the first and last character position. The first 31 characters are significant. The special character dot in an external symbol is mapped to a question mark.

Examples:

Legal	Illegal or not unique
R	REGISTER1
PART.1	REGISTER2
SORT	REGISTER3
MSL.20	MSL.3.
TOF1	TEN SPOT
TWOPI	2PI
R234567.9	R234567.

All of the identifiers in column one are legal and unique. The three identifiers beginning with REGISTER are legal but not unique for the BP since the first eight characters are the same. MSL.3. is not legal for the BP or MC68000 because of the terminating dot. TEN SPOT is not legal because of the single space between the words. The 2PI is not legal because it begins with a digit, not a letter.

2.7 CONSTANTS

THLL recognizes three kinds of constants: numbers, Boolean constants, and strings. Each is assigned a type which is used when the constant appears as an operand.

2.7.1 Numbers

Numbers fall into three categories: integers, real numbers, and scaled real constants.

2.7.1.1 Integers - There are four kinds of integers: binary, octal, decimal, and hexadecimal. Each is formed from the appropriate kind of digits and a scale part. The scale part may be omitted. If a scale part is used, it has the following form:

K [$\pm$] decimal digit(s)

and specifies a power of 2. For example, K12 means that 2**12 (=4096) is to be used as a factor of the number that has K12 as its scale part. Regardless of the kind of integer, the sequence of decimal digits following K in a scale part always specifies a decimal integer number. The possible formats for integer numbers are listed in Table 2-3.

TABLE 2-3. INTEGERS

Integer	Form
Binary	B'binary digit(s) [scale part]'
Octal	C'octal digit(s) [scale part]'
Decimal	decimal digit(s) [scale part]
Hexadecimal	X'hexadecimal digit(s) [scale part]'

Examples of integer numbers (decimal equivalent in parentheses):

Binary integers	B'101'	(5)
	B'1000K3'	(64)
	B'110101K-5'	(1)
Octal integers	C'743'	(483)
	C'3K8'	(768)
	C'21K-1'	(8)
Decimal integers	15	(15)
	395K6	(25280)
	42K-2	(10)
Hexadecimal integers	X'A4F'	(2639)
	X'1CK8'	(7168)
	X'B531K+10'	(47498240)

Note that spaces cannot appear between the characters representing an integer. Also, a plus sign or minus sign may appear only in the scale part.

An integer N, used as a literal, is assigned a type according to its magnitude in a machine-dependent manner.

BP, VAX:

Input Form	Magnitude of N	Type
Binary, octal, hexadecimal	$0 \leq N < 2**32$	Integer (I)
Decimal	$0 \leq N < 2**31$	Integer (I)
Binary, octal, hexadecimal	$2**32 \leq N < 2**64$	Double (D)
Decimal	$2**31 \leq N < 2**63$	Double (D)

MC68000:

Input Form	Magnitude of N	Type
Binary, octal, hexadecimal	$0 \leq N < 2**16$	Half (H)
Decimal	$0 \leq N < 2**15$	Half (H)
Binary, octal, hexadecimal	$2**16 \leq N < 2**32$	Integer (I)
Decimal	$2**15 \leq N < 2**31$	Integer (I)
Binary, octal, hexadecimal	$2**32 \leq N < 2**64$	Double (D)
Decimal	$2**31 \leq N < 2**63$	Double (D)

2.7.1.2 Real Numbers - Real numbers are formed from unsigned decimal integers and/or decimal fractions followed by an exponent part. The exponent part indicates a power of 10. The exponent part may be omitted. If the exponent part is used, it has the following form:

E[$\pm$] decimal digit(s)

Examples of real numbers:

```
7.      23.4986E+2
.75     .438E-05
```

As with integers, no spaces can occur between consecutive characters of the number.

All real numbers are assigned the type real, designated by R.

The range of a real number X is machine-dependent:

BP:
$.353E-9864 \le X \le .708E+9864$ with approximately 15 decimal digits of precision

MC68000:
$.353E-9864 \le X \le .708E+9864$ with approximately 9 decimal digits of precision

VAX:
$.29E-38 \le X \le 1.7E+38$ with approximately 16 decimal digits of precision

2.7.1.3 Scaled Real Numbers - A scaled real number is formed from a real number and a scale part. The real number is defined in Section 2.7.1.2, and the scale part is defined in Section 2.7.1.1.

Examples:

3.5E-3K6	(.2240)
7.02E2K-1	(351.)

The same restrictions for spaces as stated for integers and real numbers apply. Scaled real constants are of type R.

2.7.2 Boolean Constants

The Boolean constants are TRUE and FALSE. The logical values are represented by bit patterns as follows:

TRUE is represented by X'FFFFFFFF', and
FALSE is represented by X'00000000'.

The Boolean constants are assigned the type integer (I).

Any integer value used in the position of a Boolean value is interpreted as either TRUE or FALSE. 0 is interpreted as FALSE, all other integers as TRUE.

2.7.3 Strings

A string is a sequence of characters of the form:

```
#X <any sequence of characters not containing X> X
    where X is any legal character.
The maximum length of a string is 256 characters.
```

Strings are used in preset statements, as actual parameters for procedures, as right sides of assignment statements, and in FORMAT declarations. String manipulation functions are discussed in the section on standard procedures (Section 9.2).

Strings are assigned type ALPHA (A).

Examples:

```
#QTHIS IS A STRING.Q
#1IT CAN CONTAIN ANY CHARACTER:  2573, S; -A*B.1
# A.SPACE.IN.THIS.STRING.TERMINATES.THE.STRING.
```

Even a space or # can be used as the special character to begin and end the string.

2.8 COMMENT FORMS

A comment may have one of two forms:

Form 1:

```
COMMENT <any text not containing a ;> ;
```

Form 2:

```
/* <any text not including the character sequence */> */
```

A semicolon may not be used in Form 1 because it serves as the terminator of the comment. However, the /* - */ serve to bracket the comment in Form 2 so that a semicolon may be included in the text. Form 1 would appear more frequently as a preface or general description of a compile unit or procedure. Form 2 would be used to intersperse comments within the compile unit. Form 2 is also called a remark.

Example:

```
COMMENT ERRORMSG PRINTS AN ERROR MESSAGE ;

DEFINE PROCEDURE ERRORMSG(PHEAD,PBODY) ;
  POINTER PHEAD ;          /* POINTS TO AN ERROR ITEM HEADER */
  POINTER PBODY ;          /* POINTS TO AN ERROR ITEM BODY */

     BEGIN
     ...
     END ;                 /* PROCEDURE ERRORMSG */
```

It should be noted that comments are not THLL items. They can be included in a THLL compile unit to make it more readable to the user. Semantically, they are equivalent to a blank.

CHAPTER 3

DATA DECLARATIONS

The primary purpose of declarations is to provide information to the compiler about symbols used in the compile unit. In addition, the declarations document the proper usage of the symbols to the user or reviewer of a compile unit.

This chapter describes all data declarations. Symbols defined via data declarations can be used for referencing data in a compile unit. These include simple variables, arrays, stacks, components, presets, and synonyms.

An identifier which is used as a label need not be declared. All other identifiers are either predefined or must be declared via a declaration. An identifier cannot represent more than one entity in a single block.

Each declaration form is discussed in the following sections.

3.1 ALLOCATION MODE

Variables have an allocation mode which is either OWN or shared. Any variable declaration which does not contain the reserved word OWN is shared.

OWN variables are allocated statically at compile time and can be preset to initial values. Shared variables are allocated dynamically at execution time upon entrance into the block that contains their declarations. The memory locations of shared variables are not known at compile time. Therefore, shared variables cannot be preset.

If a variable should be available for use throughout a sequence of nested blocks or for examination using a debug system, it should be declared OWN.

The keyword OWN may either precede or follow the type specification in the declaration for a simple variable, array, or stack.

3.2 SIMPLE VARIABLE DECLARATION

A simple variable declaration defines an identifier to be a variable of a specific type. The following six types are available for specifying the type of a simple variable:

A. HALF - an integer quantity of a machine-dependent size:

   BP, VAX: The size is 32 bits.

   MC68000: The size is 16 bits.

B. INTEGER - a 32-bit integer quantity

C. DOUBLE - a 64-bit integer quantity

D. REAL - a floating point quantity

E. POINTER - an address quantity

F. ALPHA - a character string quantity

A declaration for a simple variable not of type ALPHA has the following form:

```
[OWN] type id1, ..., idn
```

where type is one of the keywords HALF, INTEGER, DOUBLE, REAL, or POINTER. The keyword OWN may either precede or follow the type specification in the above declaration.

A declaration for a simple variable of type ALPHA has the following form:

```
[OWN] ALPHA id1(l1), ..., idn(ln)
```

The keyword OWN may either precede or follow the type specification in the above declaration. The length of the ALPHA variable in units of characters is specified as an integer constant enclosed in parentheses following the variable name. The maximum number of characters that can be stored into the ALPHA variable is length+1. If a set of consecutively listed variables have the same length, then only the first variable in this set has to have a length specification. The variables following it are considered to have the same length until a new length is specified.

The manner in which characters are stored in an ALPHA variable is machine-dependent.

BP, MC68000:

Characters are stored into successive words, four characters per word from left to right.

VAX:

Characters are stored into successive words, four characters per word from right to left.

Strings are implemented exactly like ALPHA variables. The difference is that strings are constant and cannot be changed.

The variable specified by an identifier can only assume values of the declared type by assignment or by preset declaration. If the variable receives an assigned value, automatic type conversion occurs if the right side expression has a different type.

Examples of variable declarations:

```
INTEGER K,LOOP,RATE ;
DOUBLE OWN STATE ;
OWN REAL RANGE,LXO,BEARING ;
POINTER BUFFER.AIM.POINT ;
```

Examples of ALPHA declarations:

```
ALPHA MARS(5),M(4),N ;
ALPHA OWN TREE(40) ;
```

Note the change of length from MARS to M. Since no length is indicated for N, it has the same length as M. MARS has six characters; M and N each have five characters.

In both sets of examples, note that OWN allocation mode is included in some of the declarations.

3.3 ARRAY DECLARATION

An array declaration defines an identifier to be the name of an array. Array declarations provide information concerning the type, name, size, and allocation mode of arrays.

An array is a data structure consisting of a collection of data elements that have the same type and that can be accessed via subscripted variables. THLL provides for one-, two-, and three-dimensional arrays. Elements of these arrays are accessed via subscripted variables with one, two, or three subscripts respectively.

The following types are available for arrays:

```
HALF - an integer quantity of a machine-dependent size
       BP, MC68000: The size is 16 bits.
       VAX: The size is 32 bits.
INTEGER, DOUBLE, REAL, POINTER
```

An array declaration has the following form:

```
[OWN] type ARRAY id1(size1), ..., id(size)
```

where id1, ..., id are identifiers declared as arrays. The keyword OWN may either precede or follow the type specification in an array declaration. Each identifier is followed by a size specification in the form of one or two or three integer numbers, separated by commas and enclosed in parentheses.

If size is of this form:

```
n1,n2,n3
```

then

```
0 ... n1 is the range of the first subscript,
0 ... n2 is the range of the second subscript, and
0 ... n3 is the range of the third subscript.
```

If in an array declaration a set of consecutively listed array names has the same number of dimensions and for each dimension the same range, then only the first array name in this collection needs a size specification. For the remaining names, the same size is assumed by default until a new size is specified.

Example:

```
INTEGER ARRAY A(10),B,C,D(20,5),E ;
```

Here A, B, and C are one-dimensional arrays with 11 elements each, the subscript varying between 0 and 10; D and E are two-dimensional arrays, the first subscript varying between 0 and 20, the second between 0 and 5.

It should be noted that the size of arrays is specified in units of elements. This means two THLL words per element for REAL and DOUBLE arrays, one THLL word per element for INTEGER and POINTER arrays.

Arrays are stored in memory row by row according to the following scheme: A(i1,i2,i3) is the k'th data item from the origin A(0,0,0), where

```
k = i3 + (N3 + 1)*(i2 + (N2 + 1)*i1)
```

and N1, N2, N3 are the highest values for the first, second, and third subscripts, respectively.

3.4 STACK DECLARATION

A stack declaration defines an identifier to be the name of a stack. Stack declarations provide information concerning the type, name, size, and allocation mode of stacks.

A stack is a data structure consisting of a collection of data elements that have the same type. An element can be added to the data structure by pushing it down on the stack, the element on top of the stack can be removed by popping it off the stack. In addition, an element on the stack can be accessed as a subscripted variable with one subscript. The subscript n identifies the n'th element on the stack counting down from the top. The top element is identified by the subscript 0.

The following types are available for stacks:

```
HALF, INTEGER, DOUBLE, REAL, POINTER
```

The elements of a HALF stack have a machine-dependent size.

BP, VAX:
    The size is 32 bits.

MC68000:
    The size is 16 bits.

A stack declaration has the following form:

```
[OWN] type STACK id1(size1), ..., id(size)
```

where id1, ..., id are identifiers declared as stacks. The keyword OWN may either precede or follow the type specification of the above declaration. Each identifier is followed by a size specification in the form of one integer number enclosed in parentheses. If this number has the value n, then a maximum of n+1 data elements can be stored on the stack.

If in a stack declaration, a set of consecutively listed stack names has the same size then only the first stack name in this collection needs a size specification. For the remaining names, the same size is assumed by default until a new size is specified.

Example:

```
REAL STACK TIMES(14),COST,AZ(4) ;
```

Here, TIMES and COST are REAL stacks with a maximum of 15 elements each. AZ is a REAL stack with a maximum of five elements.

It should be noted that the size of stacks is specified in units of elements. This means two THLL words per element for REAL and DOUBLE stacks, one THLL word per element for INTEGER and POINTER stacks.

3.5 SWITCH DECLARATION

A SWITCH declaration defines an identifier to be a switch identifier. A switch identifier represents a collection of labels. Each label in this collection can be referenced in the compile unit by using the switch identifier and a subscript value.

A switch declaration has the following form:

```
SWITCH id = label-list
```

where label-list is a list of labels separated by commas.

The effect of this declaration is that the identifier following the reserved word SWITCH is associated with the sequence of labels to the right of the delimiter =. Any label listed in label-list can be referenced in the compile unit as a "subscripted label variable" of the form id(n) where n is the position number of the label in the label-list. The position number starts with 1 for the first label in label-list. If k is the position number of the last label in the list then all integer values less than 1, equal to k, or greater than k identify the last label.

Example:

```
SWITCH PATH = PAR.A,PAR.B,PAR.C,ERROR ;
```

Within the scope of PATH

```
GOTO PATH(N) ;
```

transfers control to:

```
PAR.A     if   N = 1,
PAR.B     if   N = 2,
PAR.C     if   N = 3, and
ERROR     otherwise.
```

It should be noted that the occurrence of an identifier in a switch list does not define this identifier as a label. A label is always defined by its occurrence as a label of a statement and its scope is the enclosing block in which this statement occurs. Therefore, a switch declaration must be within the scope of all labels that occur in its label-list.

The use of the SWITCH is discouraged in structured programming (see Reference 4 for further information).

3.6 COMPONENT DECLARATIONS

A component declaration defines an identifier to be the name of a component.

Components are typically used in the design of a "record" as a collection of logically related data of different types that are allocated as a group in a block of words. Any member of this group of data in the record can be accessed symbolically via a "component variable":

C(p)

where C is the component name of the member and p is a pointer value that identifies the origin of the record. The user should distinguish between the concept component, in the example C, and component variable, in the example C(p).

A component identifies a field relative to an origin. The field may be part of a word or one word or two words and is located at a fixed offset from an arbitrary origin.

Each component has four characteristics:

type of value,

offset from origin,

field size, and

sign extension.

Correspondingly, a component declaration must specify this information for a component identifier. Whenever a type keyword appears as part of a component declaration it means, with one exception, that the component value has that type. The exception is ALPHA. ALPHA components have the type INTEGER. They are explained in a separate section below.

THLL allows the user to organize the common information in the declarations of a set of component identifiers in two different ways.

3.6.1 First Form of Component Declarations

DOUBLE and REAL components are defined as follows:

```
type COMPONENT id (OFFSET n)
```

where

type is one of the keywords DOUBLE, REAL;

id is the component identifier; and

n is the offset of the double or real data element from an origin in units of THLL words.

INTEGER and POINTER components are defined as follows:

```
type COMPONENT id (sext FIELD (sb, nb), OFFSET n)
```

where

type is one of the keywords INTEGER, POINTER;

id is the component identifier;

sext is one of the keywords ARITHMETIC, LOGICAL. It can be omitted, in which case the default LOGICAL is assumed;

sb is the start bit of the field;

nb is the size of the field in bits; and

n is the offset of the field in THLL words. The offset specification can be omitted in which case the offset 0 is assumed by default.

sb and nb are integer numbers, $0 \leq sb < 32$, $0 < nb \leq 32$. They may be omitted in which case $sb = 0$, $nb = 32$ is assumed by default. The offset n is an integer number and may be negative.

The sign extension keyword ARITHMETIC means that the value of the component variable is defined as the bit pattern in the field sign extended to a full THLL word. If the leading bit in the field is 0, all bits to the left of the field in the THLL word are 0; if the leading bit of the field is 1, then all bits to the left of the field in the THLL word are 1. Hence, an arithmetic component designates a signed integer number.

The sign extension keyword LOGICAL means that the value of the component variable is defined as the bit pattern in the field extended with zeros to the left to a full THLL word. Hence, a logical component designates a logical bit pattern or an inherently positive integer quantity such as a pointer.

3.6.2 Second Form of Component Declarations

The second form of a component declaration allows the user to specify common characteristics for a list of component identifiers. A complete component declaration in this form consists of three parts:

```
type COMPONENT id-list ;
sext FIELD (sb, nb) FOR id-list ;
OFFSET n FOR id-list
```

where

type is one of the keywords INTEGER, DOUBLE, REAL, POINTER;

id-list is a list of identifiers separated by commas;

sext is one of the keywords ARITHMETIC, LOGICAL. It can be omitted, in which case the default LOGICAL is assumed;

sb is the start bit of the field;

nb is the size of the field in bits; and

n is the offset of the field in units of THLL words. The offset specification can be omitted, in which case the offset 0 is assumed by default.

Each part of the component declaration in this form defines the named characteristic for a list of identifiers. The ranges for sb, nb, and n as well as the meaning of ARITHMETIC and LOGICAL is the same in this case as for the first form of component declarations.

If this form is used for REAL or DOUBLE components, the field specification together with the sign extension should be omitted.

Examples:

The following two sets of component declarations are equivalent. Each defines four bytes BYTE1, BYTE2, BYTE3, and BYTE4 as logical bit patterns allocated from left to right in the second word after an origin.

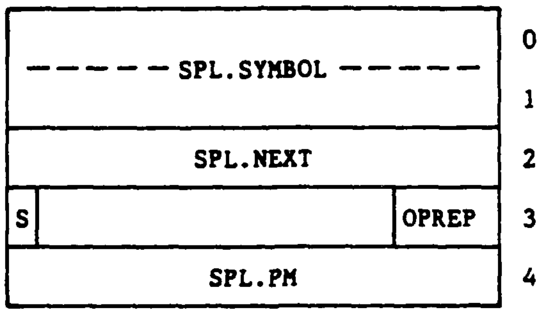
```
/* FIRST SET */
INTEGER COMPONENT BYTE1,BYTE2,BYTE3,BYTE4 ;
OFFSET 2 FOR BYTE1,BYTE2,BYTE3,BYTE4 ;
FIELD (0,8) FOR BYTE1 ;
FIELD (8,8) FOR BYTE2 ;
FIELD (16,8) FOR BYTE3 ;
FIELD (24,8) FOR BYTE4 ; /* END OF FIRST SET */

/* SECOND SET */
INTEGER COMPONENT BYTE1 (FIELD(0,8), OFFSET 2) ;
INTEGER COMPONENT BYTE2 (FIELD(8,8), OFFSET 2) ;
INTEGER COMPONENT BYTE3 (FIELD(16,8), OFFSET 2) ;
INTEGER COMPONENT BYTE4 (FIELD(24,8), OFFSET 2) ;
/* END OF SECOND SET */
```

The following set of component declarations defines a record as a group of logically related data. The example describes a simplified version of an entry into a spelling table.

```
DOUBLE COMPONENT SPL.SYMBOL (OFFSET 0) ;
        /* EIGHT CHARACTERS, LEFT JUSTIFIED, BLANK FILLED */
POINTER COMPONENT SPL.NEXT (OFFSET 2) ;
        /* POINTS TO NEXT ENTRY WITH SAME HASH ADDRESS */
INTEGER COMPONENT SPL.SYNON (FIELD (0,1), OFFSET 3) ;
        /* CURRENTLY ACTIVE SYNONYM LEFTSIDE */
INTEGER COMPONENT SPL.OPREP (FIELD (24,8), OFFSET 3) ;
        /* OPERATOR CODE, 0 FOR SYMBOLS */
POINTER COMPONENT SPL.PM (OFFSET 4) ;
        /* POINTS TO SYMBOL TABLE ENTRY REPRESENTING
           CURRENT MEANING */
```

The data structure and its components to which the above set of declarations provides symbolic access can be visualized as shown below:



S stands for SPL.SYNON and OPREP stands for SPL.OPREP. The field specification for the SPL.PM component is omitted, therefore the field is by default a full THLL word.

The entire spelling table consists of entries of this form. The subsets of the entries that have the same hash code for their names are linked together into a linked list via the pointer field SPL.NEXT. If the name is a symbol, then a pointer is provided to a symbol table record that contains all relevant information about the symbol.

3.6.3 Using Components

When a component variable is used for its value then, for REAL and DOUBLE components, the content of the component field is the value of the component variable. For INTEGER and POINTER components, the value is defined as the content of the field expanded to a one word quantity that has the type of the component. The expansion fills the new bit positions either with zeros or with the leading bit of the bit pattern in the field depending on whether the sign extension is LOGICAL or ARITHMETIC.

When a component variable is used on the left side of an assignment, the value of the right side is converted to the type of the component and stored into the component field. If the field is not a full word or a double word, then the converted value is truncated and the right-most portion that fits into the field is stored. This is the semantics of the assignment to component variables. No checks are performed if the truncation causes information to be lost. It is the user's responsibility to design the components such that their field sizes can accommodate all possible values for the particular application.

Components cannot be declared GLOBAL. If the same component declarations are to be used in several compile units, then the component declarations should be collected in an insert file and included in each compile unit where needed via the INSERT declaration (see Section 3.9).

It should be noted that a component declaration does not cause any storage allocation such as a simple variable or array declaration. The function of components is only to provide symbolic names for fields relative to an arbitrary origin.

3.6.4 Predefined Components

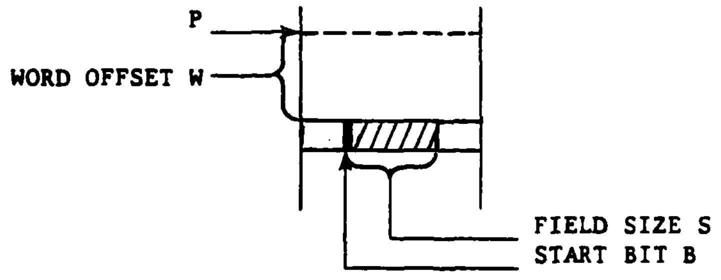
Three standard component identifiers are predefined in THLL. Each may be used without any declarations (see Table 3-1).

TABLE 3-1. PREDEFINED COMPONENTS

Name	Type	Offset	Field
WHOLEW	I	0	(0,32)
DOUBLEW	D	0	(0,64)
REALDW	R	0	(0,64)

3.6.5 Indexed Components

If C is a component and P is a pointer valued expression, then C(P) is a variable whose value is contained in the field as specified in the component declaration for C, relative to the address pointed at by P.



We call this field the component field of index 0 and picture the remaining bits of this word and the bits of all following words as one continuous bit sequence. Then the component field of index N, referred to as C(P,N), is a field of size S, starting N*S bits to the right of the field for index 0. C(P,0) is equivalent to C(P). The component variable C(P,N) is called an indexed component. N may not be negative.

To avoid word boundary problems, the start bit B and the field size S are restricted by the following conditions: 64 MOD S = 0 and B MOD S = 0. This means that S must divide both 64 and B. Hence, S can only be 1, 2, 4, 8, 16, 32, or 64.

3.6.6 ALPHA Components

For portability reasons, it is desirable to manipulate character sequences in ALPHA variables in a machine-independent manner even though these character sequences may be represented differently on different target machines. The ALPHA component serves this purpose.

The declaration:

```
ALPHA COMPONENT id (OFFSET n)
```

defines the identifier id as the name of an <u>integer</u> component with a logical field of eight bits located in the n'th word from an origin. The location within the word as well as the direction of indexing is machine-dependent.

BP, MC68000:
The field is the left-most byte in the word and indexing proceeds to the right as for all other indexed components.

VAX:
The field is the right-most byte in the word and indexing proceeds to the left. This is different from all other indexed components on the VAX but it is in accordance with the storage of characters in an ALPHA variable on the VAX.

ALPHA component variables can only be used as indexed component variables. This means that the user should always write:

```
CA(p,i)
```

when referencing an ALPHA component CA, even when the index i is 0.

3.7 PRESET DECLARATION

The preset declaration is used to initialize OWN variables at compile time. Simple, subscripted, and component variables may be preset. However, subscripted variables using a stack identifier cannot be preset.

3.7.1 Syntax of Presets

Two forms of preset declarations are provided, a simple preset declaration and a compound preset declaration.

A. A simple preset declaration has the form:

```
PRESET     preset-statement
```

B. A compound preset declaration has the form:

```
PRESET
     BEGIN
     preset-statement ;
        .
        .
        .
     preset-statement ;
     END
```

Two kinds of preset-statements are allowed:

A. var = e1, ..., ek

B. var TO var = e1, ..., ek

In these statements, var can be a simple variable (SV), a subscripted variable using an array identifier (ASUB), a component variable (COMP), or an indexed component variable (ICOMP). Each expression ei must be evaluatable at compile time and must not be an assignment expression.

3.7.2 Semantics of Presets

The left side of a preset statement always specifies the OWN storage area for a symbol or a part of it. Presetting takes place in this storage area only.

If the left side is an SV or an ASUB, then the symbol is the simple variable id or the array id, respectively. If the left side is a COMP or ICOMP, say C(pe,n), then the symbol is the symbol specified by the value of the pointer expression pe; e.g., in the simplest case pe may be LOC B(0), where B is a one-dimensional OWN array, or pe may be a pointer that was previously preset to LOC B(0). A pointer that is used for presetting a component must always be relative to a THLL symbol. Presetting of a component takes place in the storage area for this symbol.

If the left side is a range var1 to var2, then the symbol specified by var1 and by var2 must be the same and presetting takes place in the storage area for this symbol only.

Let S be the symbol specified by the left side. A storage element is defined to mean one halfword, one word, or one doubleword according to the following table:

Type of left side	Machine	Size of storage element
HALF	MC68000	16 bits
	VAX	32 bits
	BP	16 bits for subscripted variables 32 bits for other variables
INTEGER, POINTER	any	32 bits
DOUBLE, REAL	any	64 bits

Let NE be the number of storage elements left in the storage area for S, starting at the point specified by the left side. Similarly, let NF be the number of fields left in the storage area for S if the left side is an ICOMP. The field size is as defined in the component declaration. If the left side is a range, let NR be the number of elements (for ASUB and COMP) or the number of fields (for ICOMP) in the range. Then the above preset statements (1) and (2) have the following meaning:

left side = e1, ..., ek

Left Side	Meaning
SV, type ALPHA	The ALPHA variable is preset to e1. Only one right side element e1 is used.
SV, other types	One storage element is preset to e1. Only one right side element e1 is used.
ASUB	i = min(k,NE) successive storage elements are fully preset to e1, ..., ei.
COMP	The same field in i = min(k,NE) successive storage elements is preset to e1, ..., ei (vertical sequence).
ICOMP	i = min(k,NF) successive fields are preset to e1, ..., ei (horizontal sequence).

ASUB TO ASUB	NR successive storage elements are preset to e1, ..., eNR if NR ≤ k, to e1, ..., ek, ek, ..., ek otherwise.
COMP TO COMP	The same field in NR successive storage elements is preset to e1, ..., eNR if NR ≤ k, to e1, ..., ek, ek, ..., ek otherwise (vertical sequence).
ICOMP TO ICOMP	NR successive fields are preset to e1, ..., eNR if NR ≤ k, to e1, ..., ek, ek, ..., ek otherwise (horizontal sequence).

In each case, if the number of preset expressions on the right side is greater than the number of storage elements or fields in the storage area for the symbol specified by the left side, then an error message is issued. Also, presetting of fields or elements more than once and presetting of overlapping fields or elements are not allowed.

It should be noted that the semantics of presets for component variables on the left side depends on whether the component variable is indexed or not. If not indexed, then the next field to be preset is the field, as defined for the component, in the next storage element of the data area; if indexed then the next field to be preset is the adjacent field that has the size of the component. This distinction is also made when the index of the component variable is 0.

ALPHA component variables must always be indexed. This means that a simple preset statement with an ALPHA component variable on the left side always presets a sequence of adjacent fields if more than one field is preset. The case in which the same field in a sequence of array elements is preset cannot arise.

Presetting of simple and subscripted variables can always be accomplished using component variables on the left side. The following table lists the different cases and the semantically equivalent replacements of simple and subscripted variables by component variables.

```
INTEGER COMPONENT CI  (FIELD (0,32), OFFSET 0) ;
DOUBLE COMPONENT  CD  (OFFSET 0) ;
REAL COMPONENT    CR  (OFFSET 0) ;
POINTER COMPONENT CP  (FIELD (0,32), OFFSET   ;
INTEGER COMPONENT CLH (FIELD (0,16), OFFSET 0) ;
INTEGER COMPONENT CRH (FIELD (16,16), OFFSET 0) ;
```

Left side variable X	Replaceable by
SV, ASUB type I	CI(LOC X)
SV, ASUB type D	CD(LOC X)
SV, ASUB type R	CR(LOC X)
SV, ASUB type P	CP(LOC X)
ASUB type H	This case is machine-dependent: BP: CLH(LOC X) if X specifies the left half of a THLL word, CRH(LOC X) if X specifies the right half of a THLL word. MC68000: CLH(LOC X) VAX: CI(LOC X)
SV type H	This case is machine-dependent: BP, VAX: CRH(LOC X) MC68000: CLH(LOC X)

Conversely, presets using component variables on the left side cannot, in general, be accomplished using only simple and subscripted variables.

Examples:

```
OWN HALF Y1,Y2,Y3 ;
OWN HALF ARRAY A1(20),A2 ;
OWN INTEGER ARRAY B(5),B1,C,D,E,F,G,I,K,L ;
INTEGER COMPONENT CI  (FIELD  (0,32), OFFSET 0) ;
INTEGER COMPONENT CLH (FIELD  (0,16), OFFSET 0) ;
INTEGER COMPONENT CRH (FIELD (16,16), OFFSET 0) ;
DOUBLE COMPONENT  CD  (OFFSET 0) ;
ALPHA COMPONENT   CA  (OFFSET 0) ;
```

```
PRESET
     BEGIN
     A1(0) = 10,20,30 ;              /* BP,MC68000: First 3 halfwords
                                        are set to 10,20,30
                                        VAX: First 3 words are set
                                        to 10,20,30 */
     A2(0) TO A2(12) = 10,20,30 ;    /* BP,MC68000: First 13 halfwords
                                        are set to 10,20,30,...,30
                                        VAX: First 13 words are set
                                        to 10,20,30,...,30 */
     CLH(LOC A1(5)) = 10 ;           /* BP: A1(4) is set to 10
                                        MC68000: A1(5) is set to 10
                                        VAX: Left half of A1(5)
                                        is set to 10 */
     CRH(LOC A1(7)) = 10 ;           /* BP: A1(7) is set to 10
                                        MC68000: A1(8) is set to 10
                                        VAX: Right half of A1(7)
                                        is set to 10 */
     CI(LOC A1(12)) = 10 ;           /* BP,MC68000: A1(12) is set to 0,
                                        A1(13) is set to 10
                                        VAX: A1(12) is set to 10 */
     CI(LOC A1(15)) = 10 ;           /* BP: A1(14) is set to 0,
                                        A1(15) is set to 10
                                        MC68000: A1(15) is set to 0,
                                        A1(16) is set to 10
                                        VAX: A1(15) is set to 10 */
     CI(LOC B(0)) = 10,20,30 ;       /* First 3 words
                                        are set to 10,20,30 */
     CI(LOC B1(0),0) = 10,20,30 ;    /* First 3 words
                                        are set to 10,20,30 */
     CLH(LOC C(0)) = 10,20,30 ;      /* Left half of first 3 words
                                        is set to 10,20,30 */
     CLH(LOC D(0),0) = 10,20,30 ;    /* First 3 halfwords are set
                                        to 10,20,30 */
     CRH(LOC E(0)) = 10,20,30 ;      /* Right half of first 3 words
                                        is set to 10,20,30 */
     CRH(LOC F(0),0) = 10,20,30 ;    /* Starting with second half-
                                        word, 3 halfwords are set
                                        to 10,20,30 */
     CD(LOC G(0)) = 10,20,30 ;       /* First 3 doublewords are set
                                        to 10,20 30 */
     CD(LOC I(0),0) = 10,20,30 ;     /* First 3 doublewords are set
                                        to 10,20 30 */
     Y1 = 10 ;                       /* Y1 is set to 10 */
     CLH(LOC Y2) = 10 ;              /* BP,VAX: left half of Y2
                                        is set to 10
                                        MC68000: Y2 is set to 10 */
     CRH(LOC Y3) = 10 ;              /* BP,VAX:  right half of Y3 is
                                        set to 10
                                        MC68000: illegal, field outside
                                        of specified data area (Y3) */
```

```
CA(LOC K(0),0) = X'41',X'42',X'43' ;
                                /* 3 bytes (first, second, third)
                                   in K(0) are set to character
                                   code for A,B,C */
CA(LOC L(1),4) TO CA(LOC L(1), 9) = X'41',X'42',X'43' ;
                                /* 6 bytes (fifth, ..., tenth) are
                                   set to character code for
                                   A,B,C,C,C,C */
/* The meaning of "first", "second". ... for
indexed ALPHA components is machine-dependent.
See section on ALPHA components */
END ;                            /* end of compound PRESET */
```

3.7.3 Compile Time Expressions

A compile time expression is an expression for which a value can be computed at compile time. A compile time expression is normally used on the right side of preset statements. When used in executable statements, the value is still computed at compile time and used but not recomputed at runtime.

A compile time expression is:

A. A constant,

B. A previously preset variable,

C. LOC V where V is a format identifier or OWN variable,

D. An expression of the form:

OP1 e

or

e1 OP2 e2

where e, e1, and e2 are compile time expressions;

and OP1 belongs to the set {+, -, LOC, NOTB};

OP2 belongs to the set {+, -, *, /, MOD, ORB, XORB, ANDB}.

For compile time expressions, the same type rules, including type conversion, apply as for arbitrary expressions. These rules are specified in the type tables, Appendix B. One exception is that expressions of the form LOC F1 - LOC F2, where F1, F2 are format identifiers, are not allowed. Another exception is that real operands are not supported except for unary plus and unary minus.

3.7.4 Preset Expressions (SWA, LINKWORD, INITWORD)

A preset expression is an expression that can occur on the right side of a preset statement.

A preset expression is a compile time expression or a functional expression of the form f(e1, ..., en) where f is a procedure identifier and each ei, i=1, ..., n is a compile time expression or is LOC P, P being a procedure identifier.

Three functional expressions are supported for preset expressions:

```
SWA(N) (see Section 9.5)
LINKWORD(LOC P,N)
INITWORD(LOC P,N)
```

where N is a compile time integer expression and P is a procedure identifier which must be GLOBAL or EXTERNAL.

The meaning of LINKWORD and INITWORD is machine-dependent:

VAX, MC68000:

The value of LINKWORD and INITWORD is an integer containing the the address of the procedure P.

BP:

The value of LINKWORD(LOC P,N) is an integer representing a linkage word which is used by the ENTER TASK instruction when procedure P is invoked. LINKWORD(LOC P,N) should not be used as an operand except for the assignment operator (=) in a preset.

```
PRESET X = LINKWORD(LOC P,N):
```

is compiled to:

```
LABX DATA,2  .LTO(P) N
```

where LABX is a label in the data area corresponding to X. After being processed by the LINKER, this word contains:

LTO	LTA

where

LTO is the linkage table offset for P

and

LTA (= N) the virtual address of the linkage table.

It is recommended that the following synonyms for N be used:

```
SYNONYM LTA.M=X'D019' ; /* LTA FOR MONITOR */
SYNONYM LTA.E=X'D219' ; /* LTA FOR EXEC */
SYNONYM LTA.T=X'F019' ; /* LTA FOR TASK */
```

The value of INITWORD is an integer containing the information for a dedicated interrupt cell. The procedure identifier P in the first argument LOC P must be a GLOBAL or EXTERNAL EXEC procedure. N represents the upper half of the Program Status Word (PSW) (Reference 5) when P is to be activated. INITWORD(LOC P,N) should only be used as the right-hand side of a preset assignment statement.

3.8 SYNONYM DECLARATION

A synonym declaration defines a THLL item to be the name of a piece of THLL text. When this name is used within the scope of the synonym declaration then the corresponding THLL text is inserted. Several synonym definitions can be combined into one compound synonym declaration.

A synonym declaration consists of the keyword SYNONYM followed by one synonym element or followed by a sequence of synonym elements separated by semicolons and enclosed in BEGIN-END brackets. The syntax for a synonym element is as follows:

```
syn-left = syn-right
```

where syn-left is any THLL item other than a constant. Such a THLL item is referred to as a non-const. The right side, syn-right, of a synonym element has one of the following forms:

```
constant
non-const THLL-text non-const
```

Here constant is any of the THLL constants described in Section 2.7. THLL-text is an arbitrary piece of THLL text that does not contain the THLL item non-const which starts and terminates the synonym right side. In the first case, the right side is the constant; in the second case, it is the THLL-text that is being named by the left side of the synonym.

It should be noted that a negative number such as -24 is not one THLL item, it is a sequence of two THLL items.

By default, a synonym right side cannot exceed 1500 items. This limit can be increased by the SYN directive (see Appendix C).

The scope of a synonym declaration begins with the item following the semicolon that terminates the synonym definition and ends at the end of the block containing the definition or at the beginning of a new synonym definition with the same left side, whichever comes first. The effect of a synonym declaration within its scope is that every occurrence of the synonym left side of a synonym element outside a synonym declaration is textually replaced in the source text by the corresponding synonym right side. After the replacement, scanning starts with the first item of the item sequence replacing the identifier. Therefore, the right side of a synonym element can contain another synonym left side which is also expanded to its corresponding item sequence. The synonym must be declared before its first usage.

Examples:

A. Synonym declaration

```
SYNONYM PI = 3.141592 ;

SYNONYM
     BEGIN
     MASK1 = X'00000001' ;
     MASK2 = X'0000FFFF' ;
     MASK3 = X'FFFF0000' ;
     END ;

SYNONYM FUNC = $ COS(PI*A) + SIN(PI*B) $ ;

SYNONYM EXCHIK = $ BEGIN
                   INTEGER J ;
                   J = I ;
                   I = K ;
                   K = J ; /* EXCHANGE I AND K */
                   END $ ;
```

```
SYNONYM RIGHTHALF = $ MASK2 ANDB $ ;
                     /* TO BE USED AS UNARY OPERATOR, E.G.
                        RIGHTHALF X */

SYNONYM LEFTHALF = $ MASK3 ANDB $ ;
                    /* TO BE USED AS UNARY OPERATOR */
```

B. Use of synonyms:

```
MASK1 expands to X'00000001'

EXCHIK expands to BEGIN
                  INTEGER J ;
                  J = I ;
                  I = K ;
                  K = J ; /* EXCHANGE I AND K */
                  END

RIGHTHALF X expands to X'0000FFFF' ANDB X

LEFTHALF X expands to X'FFFF0000' ANDB X
```

3.9 INSERT DECLARATION

The INSERT declaration permits the programmer to include source text from other than the current compiler source. The source text to be included is specified as a file from a directory in the following form:

```
INSERT fname(dname)
```

where fname is a filename with the extension .THI and dname is a directory name or a VAX logical name used to specify a directory. Appendix G describes how the directory is located.

INSERT files normally contain a set of logically related declarations that are to be included in many different compile units. This feature permits the programmer to maintain a single copy of these declarations. In general, however, an INSERT file may contain any piece of source text. The effect of the INSERT declaration is that the character sequence "INSERT fname(dname)"; is replaced by the content of the INSERT file.

An INSERT declaration is not a declaration in the sense that it defines the meaning of a symbol. It could more accurately be called a directive since it instructs the compiler to read source text from a new file. Normally, an INSERT declaration occurs in a declaration position and contains a sequence of declarations. This is the reason for the misnomer "INSERT declaration."

CHAPTER 4

EXPRESSIONS

An expression is a rule for computing a new value from existing values. Expressions are built from constants and variables using operators, valued procedures (functions), and control structures. Expressions are used in constructing algorithms.

All operators that can be used in expressions are described in Section 2.5.1. They are predefined in the language. The user has no facility to define operators. A set of standard procedures is predefined (Chapter 9). In addition, THLL allows the user to define procedures (Chapter 6). Two control structures are provided that can be used to form expressions: the if construct for conditional expressions (Section 4.6) and the CASE construct for case expressions (Section 4.7).

Every expression has a type. The meanings of the six different types provided by THLL are as follows:

HALF - an integer quantity of a machine-dependent size

BP: 16-bit quantity if the expression is a subscripted array variable, 32-bit quantity otherwise;

MC68000: 16-bit quantity;

VAX: 32-bit quantity.

INTEGER - a 32-bit integer quantity

DOUBLE - a 64-bit integer quantity

REAL - a floating point quantity

POINTER - an address quantity

ALPHA - a character string quantity

It should be noted that the result of an operation involving subscripted variables is not a subscripted variable. For example, if A is a HALF array, then on the BP A(1) is a 16-bit quantity, but the value -A(1) is represented as a 32-bit quantity.

4.1 PRIMARY OPERANDS

Constants, that is, numbers, Boolean (logical) values, and strings as defined in Section 2.7, and variables are basic elements. Basic elements can serve as primary (lowest level) operands. Other constructs, even though they are of a composite structure, can also be used as primary operands. These constructs are:

Address expressions using LOC and LOCA,

Functional expressions,

Conditional expressions, and

Case expressions.

Any expression enclosed in parentheses becomes a primary operand. These kinds of expressions listed above are discussed in the following sections.

4.2 VARIABLES

There are three kinds of variables: simple, subscripted, and component. A variable names a value which may be changed through the use of an assignment. Each variable has a type associated with it as indicated by the data declaration (see Chapter 3). The possible types are: half (H), integer (I), double (D), real (R), pointer (P), and ALPHA (A).

4.2.1 Simple Variables

A simple variable is represented by an identifier of type H, I, D, R, P, or A.

4.2.2 Subscripted Variables

A subscripted variable is represented by an identifier followed by a sequence of subscript expressions enclosed by parentheses. Subscripted variables refer to elements of arrays of type H, I, D, R, P, or to elements of stacks of type H, I, D, R, or P. A maximum of three subscript expressions is allowed for array subscripted variables. Stack subscripted variables have only one subscript expression.

4.2.3 Component Variables

A component variable is represented by a component identifier followed by one or two expressions enclosed in parentheses. The first expression must be of type P and specifies an origin relative to which the component is defined. The second expression, if present, must be of type I and specifies an index. Component variables may be of type I, D, R, or P.

4.2.4 Examples

Simple	Subscripted	Component
A1	VG(3)	MSL(J+3)
AVERAGE	W(I,7)	ITEM(PART)
TR.29	REC(L, W, H)	CAT(LX+4-X)

Additional information concerning variables can be found in the sections on declarations.

4.3 FUNCTIONS

A functional expression (function call) in THLL is designated by an identifier followed by parentheses enclosing a list of parameters. It has the following form:

```
pid (parameter-list)
```

where pid is a procedure identifier and parameter-list is a sequence of actual parameters separated by commas. A call to a function without parameters consists only of the function name:

```
pid
```

An item in the parameter list can be an expression, an array identifier, a stack identifier, a device identifier, ENTRYP of a procedure identifier, or a format identifier.

A function call is the application of a procedure to a fixed set of parameters resulting in a value. A function may have a type associated with it as indicated in the procedure declaration. It must be of type H, I, D, R, or P (see Section 6.2). An unvalued function is said to be of type N (no type).

There is a set of predefined (standard) functions for which the user does not have to supply a declaration. Details of these functions are described in Chapter 9.

4.4 SIMPLE EXPRESSIONS

Simple expressions are primary operands (Section 4.1) or are constructed from them using the bit, arithmetic, relational, and logical operators (Section 2.5.1). Expressions are evaluated according to the order of precedence of the operators as given in Table 4-1, in general from left to right. Sometimes optimization techniques require rearrangement of subexpressions. Therefore, the programmer can only assume that the order of evaluation is compatible with the precedence table. No specific order of evaluation can be guaranteed. This should not be viewed as any problem. The only thing a user must keep in mind is to not write expressions whose value depends on a particular order of evaluating subexpressions permitted by the precedence rules.

TABLE 4-1. OPERATOR PRECEDENCE

Precedence	Operator
0 (highest)	LOC, LOCA
1	NOTB
2	ANDB
3	ORB, XORB
4	**
5	*, /, MOD
6	unary−, unary+
7	+, -
8	LES, LEQ, EQL, GEQ, GRT, NEQ
9	NOT
10	AND
11	OR, XOR
12 (lowest)	=

Within the same precedence class, evaluation proceeds normally from left to right. For example, A+B+C is the same as (A+B) + C. The only exception is the assignment operator (=) where evaluation is right to left. For example, X=Y=Z=A+B is the same as X=(Y=(Z=A+B)).

When the logical operators are used, it may not be necessary to evaluate both operands. The programmer should avoid using expressions with side effects, such as assignments and procedure calls, as operands. The evaluation of the operators AND and OR is done as follows:

```
X AND Y  :  IF X EQL FALSE THEN X ELSE Y IFEND

X OR Y   :  IF X EQL TRUE THEN X ELSE Y IFEND
```

Every expression has a type which is determined from the operand types and the value resulting from each operation. For example: let the expression be A+B where A is of type I and B is of type R. The result of the addition produces a value of type R. Appendix B contains the matrices which indicate the types of values resulting from the various types of operands for each operator.

Examples of expressions using logical or arithmetic operators:

```
Y GRT V OR Z LES Q
NOT (A OR B) AND C
-M*(ALT MOD T)/Z
MSL(I) XORB MSL(K)
PHI*TG/RS**2
```

4.5 ASSIGNMENT EXPRESSIONS

Assignment expressions are always of the form:

```
variable = expression
```

Since the right side expression can, in particular, be an assignment expression, multiple assignments of the form:

```
var1 = var2 = var3 = ... = vark = expression
```

are allowed in THLL. The left to right rule for evaluation does not apply to assignment expressions. The rules for evaluating assignment expressions are described below:

Examples:

```
A. RT = SIN (B) * COF1

B. A(I+1, 2) = - 1

C. M = N = ARK = 7.5/SUM

D. LINE1 = #X MSL.NO.---STATUS X
```

The following steps are applied in evaluating assignments.

1. If the variable on the left is subscripted, the subscript expressions are evaluated. (Note Example B.)

2. If the variable on the left is a component one, the component expression is evaluated.

3. The expression on the right is evaluated. This is the value of the assignment expression.

4. The value from 3 is converted to the type of the left-hand variable.

5. Assignment of the converted value takes place.

In Example C, assume that M and ARK are real, N and SUM are integer and that 3 is the value of SUM. The following table describes the sequence of evaluations:

Expression	Value	Type
7.5/SUM	2.5	REAL
ARK = 7.5/SUM	2.5	REAL
ARK	2.5	REAL
N = ARK = 7.5/SUM	2.5	REAL
N	2	INTEGER
M = N = ARK = 7.5/SUM	2.5	REAL
M	2.5	REAL

In the last assignment, the unconverted value of the right side is assigned to the variable M.

When a component variable specifying a field of length K appears on the left, only the K right-most bits of the value of the right-hand expression are moved into the field.

The type conversions of step 4 are performed according to Table 4-2. The row headings indicate the variable type and the column headings indicate the expression type.

The assignment of a string to an ALPHA variable copies the string on the right into the variable on the left which has been declared type A. An error message indicates when the string is truncated if all of it does not fit into the ALPHA variable on the left.

TABLE 4-2. TYPE CONVERSION

Left-Hand =	Right-Hand H	I	D	R	P	A
H	H	H	H	H	*	*
I	I	I	I	I	*	*
D	D	D	D	D	*	*
R	R	R	R	R	*	*
P	*	*	*	*	p	*
A	*	*	*	*	*	A

The assignment of a string to a double variable copies the first eight characters from the string to the double variable. The string is considered left-justified and blank-filled in the variable. Only an ALPHA string can be used on the right of the assignment, not an ALPHA variable.

4.6 CONDITIONAL EXPRESSIONS

Conditional expressions have the form:

```
IF expl THEN
     exp2
[ELSE
     exp3]
ENDIF
```

where expl, exp2, and exp3 are expressions. The enclosing IF - ENDIF brackets are required. The ELSE clause is optional.

There may be several paired expressions separated by THEN in a conditional statement. In this case, the pairs are separated by the delimiter ELSEIF.

Consider:

```
IF P1 THEN
     e1
ELSEIF P2 THEN
     e2
     .
     .
     .
ELSEIF Pn THEN
     en
[ELSE
     ep]
ENDIF
```

To evaluate the conditional expression, each p1, ..., pn is evaluated in turn until <u>one</u> of the pi gives a value TRUE. Thus, not every pi is necessarily evaluated. Then the corresponding ei is evaluated and becomes the value of the conditional expression. If no pi is true, the conditional expression evaluates to the value of the expression following the ELSE or to the default value (as defined below) of the common type of e1, ..., en. Because the pi are interpreted as logical values, they cannot be of type R, P, or A. A zero value for pi is FALSE; a nonzero value is TRUE.

The type of the conditional expression is the lowest inclusive common type of the ei, which is defined below.

Let T1, ..., Tn be the types of e1, ..., en. Let PAIR be a type function defined by Table 4-3. Then the type of e1, ..., en is

$$\text{TYPE (T1, ..., Tn)} = \begin{cases} \text{T1 if n = 1} \\ \text{PAIR (TYPE(T1, ..., Tn-1), Tn)} \\ \text{otherwise.} \end{cases}$$

TABLE 4-3. TYPE OF A SET OF EXPRESSIONS

		expression ei						
	PAIR	H	I	D	R	P	A	N
	H	H	I	D	R	N	N	N
	I	I	I	D	R	N	N	N
	D	D	D	D	R	N	N	N
expression ej	R	R	R	R	R	N	N	N
	P	N	N	N	N	P	N	N
	A	N	N	N	N	N	A	N
	N	N	N	N	N	N	N	N

The following table defines for each type a default value:

Type	Default Value
H	0 of type H
I	0 of type I
D	0 of type D
R	0.0 of type R
P	0 of type P
A	#// (null string)

4.7 CASE EXPRESSIONS

The general form for a case expression is:

```
CASE p DO
     e1
NEXTCASE
     e2
  .
  .
  .
NEXTCASE
     en
ENDCASE
```

The expression p is evaluated to an integer (if p is real, it is converted) which serves as an index to select which expression ei is done. If p produces a value less than 1 or greater than n, then p is assigned the value of n. The programmer must keep this in mind if the limits of p are not known or if an out of bounds value is to be processed as an error.

The type of the case expression is the lowest inclusive common type of the ei expressions as defined in Section 4.6.

CHAPTER 5

STATEMENTS

The primary tools for coding algorithms are expressions and statements. The difference between them is that expressions have values while statements produce only effects on the environment. Since statements have no value, they are assigned the type N for no type.

Statements can be divided into two categories:

A. Proper statements and

B. Change of control statements.

These categories are subdivided in the sections below.

5.1 LABELED STATEMENTS

A labeled statement is a statement or expression preceded by an identifier and a colon.

```
id: statement/expression
```

More generally, a sequence of identifiers can be used, each one followed by a colon.

```
id1: id2: ... idk: statement/expression
```

Each one of these identifiers can serve to identify this statement. This facility is needed for GOTO, EXIT, and LOOPEXIT statements. It is not true that a labeled statement can occur wherever a statement can occur. The statement representing the body of a loop or the alternative in a conditional statement or in a case statement must not be a labeled statement.

5.2 PROPER STATEMENTS

A proper statement is a block or one of the following statements: compound, conditional, case, loop, or null.

5.2.1 Blocks and Compound Statements

A block may be defined as a sequence of zero or more declarations followed by one or more statements or expressions, all separated by semicolons (;), and embraced by BEGIN-END brackets. The structure of a block can be indicated by indentation and alignment of the brackets as shown below:

```
BEGIN
dl ;
.
.
.
dn ;
el ;
.
.
.
em ;
END
```

where the dl to dn are declarations, and the el to em are expressions or statements.

Blocks may be nested, and this can be indicated by further levels of indentation (see the examples in Chapter 10).

A compound statement is a block which contains no declarations.

Each block introduces a new level of nomenclature. All identifiers with the exception of labels are defined through declarations at the beginning of the block. These identifiers are local to the block in which they are defined. Hence, the entity represented by the identifier inside the block does not exist outside the block; and any entity represented by the same identifier outside the block cannot be accessed from within the block.

Any identifier referenced in a block, but not declared there, is said to be nonlocal to that block. This type of identifier represents the same entity in the block and all embracing blocks up to the one in which it is defined.

An expression is said to be in a statement position if it follows BEGIN or a semicolon or a colon.

5.2.2 Conditional Statement

The conditional expression:

```
IF p1 THEN
     e1
ELSEIF p2 THEN
     e2
     .
     .
     .
ELSEIF pj THEN
     ej
ELSE
     ek
ENDIF
```

becomes a conditional statement if the set of e1, e2, ..., ek, which can be statements or expressions has type N. The type of the set is determined as described in Section 4.6. If at least one of the ei is a statement, the conditional expression becomes a statement. Each ei must be unlabeled.

5.2.3 Case Statement

The case expression:

```
CASE e DO
     e1
NEXTCASE
     e2
     .
     .
     .
NEXTCASE
     ek
ENDCASE
```

becomes a case statement if the type of the set of e1, ..., ek, which can be statements or expressions, is type N. The type of the set e1, ..., ek is determined as described in Section 4.6. Each ei must be unlabeled.

Sections 4.6 and 4.7 discuss conditional and case expressions. Examples can be found in Chapter 10.

5.2.4 Loop Statements

There are four types of loop statements. Each type uses the reserved word DO followed by an expression or a proper statement s terminated by the keyword ENDDO. s is called the loop body and is evaluated zero or more times according to conditions. The loop body must not be labeled.

In all forms to be described, e must be of type H, I, or D since it is interpreted as true or false. The expressions e1, e2, and e3 may be of any type. The only restrictions are those resulting from the use of the type of conversion rules for =, +, -, and SIGN.

In Forms 2-4, v must be a simple variable. Examples for each type are given.

Form 1:

```
WHILE e DO
     s
ENDDO
```

For this form, e is evaluated. If a true (nonzero) value results, s is evaluated and control returns to evaluate e again. Otherwise, the loop is complete.

Example 1:

```
I1 = 0 ;
WHILE I1 LES 10 DO
     I1 = I1 + 1
ENDDO
```

When this statement completes, the value of I1 is 10.

Form 2:

```
FOR v = e1 STEP e2 WHILE e DO
     s
ENDDO
```

The expressions e1 and e2 are evaluated and v is assigned the value of e1. Next e is evaluated. If a true (nonzero) value results, s is evaluated and v is incremented by the value of e2. Control returns to the point of evaluating e again. Otherwise, the loop is complete.

Example 2:

```
I = 20 ;
FOR C = 0 STEP 1 WHILE I GRT 12 DO
     I = I - 1
ENDDO
```

Upon completion of this loop, I is 12 and C is 8.

<u>Form 3</u>:

```
FOR v = e1 STEP e2 UNTIL e3 DO
     s
ENDDO
```

This statement is equivalent to a variation of Form 2.

```
v3 = e3 ;
FOR v = e1 STEP e2 WHILE (v-v3)*SIGN(e2) LEQ 0 DO
     s
ENDDO
```

Example 3:

```
C = 0 ;
FOR I = 0 STEP 1 UNTIL 10 DO
     C = C + 1
ENDDO
```

At the end of the loop, both I and C have the value 11.

<u>Form 4</u>:

```
FOR v = e1 REPEAT e2 WHILE e DO
     s
ENDDO
```

The expression e1 is evaluated and the resulting value assigned to v. Next e is evaluated. If e returns a true (nonzero) value, s is evaluated and e2 evaluated. The value of e2 becomes the new value for v and control returns to the point of evaluating e. If e is false (zero), the loop is complete. Note that in this form v is never incremented as in Forms 2 and 3; instead it is assigned the value of e2 each time through the loop.

Example 4:

```
C = 14 ;
FOR I = 1 REPEAT I*2 WHILE C GRT 10 DO
     C = C - 1
ENDDO
```

When this loop completes, I has the value 16 and C has the value 10.

5.2.5 <u>Null Statement</u>

The null statement is represented by the reserved word NULL and specifies a no operation. It could be used to fall to the end of a conditional or case statement or at any point where a no operation is appropriate.

5.3 CHANGE OF CONTROL STATEMENTS

There are three kinds of change of control statements: (1) <u>GOTO statements</u>, (2) <u>EXIT statements</u>, and (3) <u>RETURN statements</u>. Each are discussed in turn.

5.3.1 <u>GOTO Statement</u>

GOTO statements have the following form:

$$\text{GOTO} \begin{cases} \text{label identifier} \\ \text{switch identifier (subscript)} \end{cases}$$

The obvious effect is that control is diverted to the statement identified by the specified label or switch.

It should be noted that both forms of the GOTO statement are, in general, considered unacceptable in structured programming. The second form can be useful for selected case statements as explained in Reference 4.

5.3.2 <u>EXIT Statement</u>

An EXIT statement has one of the following forms:

EXIT [label identifier]

LOOPEXIT [label *identifier*]

The effect is to transfer control to:

A. The end of the present block (EXIT),

B. The end of an embracing block labeled by the label identifier (EXIT label),

C. The end of the present loop (LOOPEXIT), and

D. The end of an embracing loop labeled by the label identifier (LOOPEXIT label).

Control is not transferred to the statement on which the label appears.

EXIT is used to terminate execution of a block of code. If the labeled EXIT form is used, the label must appear on the BEGIN of the block to be terminated.

Example A: Unlabeled EXIT.

```
BEGIN
S = SIN(PHIO) ;
ERR = 0 ;
C = COS(PHIO) ;
     BEGIN
     X = R * S ;
     IF X THEN
          EXIT
     ENDIF ;
     ERR = 5 ;
     END ;
Y = 1.0 + CF1 * X + CF2 * X**2 ;
END ;
```

Here the inner block is the one which terminates if X is nonzero and ERR remains set to zero.

Example B: Labeled EXIT.

```
L1: BEGIN
    S = SIN(PHIO) ;
    C = COS(PHIO) ;
        BEGIN
        X = R * S ;
        IF X = 0.0 THEN
            EXIT L1
        ENDIF ;
        END ;
    Y = 1.0 + CF1 * X + CF2 * X**2 ;
    END ;
```

In this example, the outer block labeled L1 is terminated if X is zero. In this case, Y is not evaluated.

If EXIT is used within a block which is the body of a loop statement, then this block is exited and the next iteration of the loop is executed (if any). Care must be exercised in placing the label if one is used since the loop body cannot be labeled. In this case, the loop body should be made an unlabeled block containing one labeled statement:

Example C: Labeled EXIT within a Loop.

```
FOR K = 0 STEP 1 UNTIL 23 DO
    BEGIN
    L1: BEGIN
        E = SIN(AL(K)) * COS(DL(K)) ;
        IF E LEQ MIN THEN
            EXIT L1
        ENDIF ;
        F = K2 / E ;
        END ;
    END
ENDDO
```

In Example C, the body of the loop statement is executed 24 times. However, F may not be evaluated on each iteration. The evaluation of F takes place on those iterations where E exceeds the value of MIN. In this example, the label could be removed and the result would be unchanged. But suppose the simple IF statement were replaced by this compound IF statement.

```
FOR K = 0 STEP 1 UNTIL 23  DO
     BEGIN
     L1:  BEGIN
          IF E LEQ MIN THEN
               BEGIN
               ERR = 1 ;
               EXIT ;
               END
           ENDIF ;
           F = K2 / E ;
           END
     END
ENDDO
```

Omitting the label in this modified example would permit the evaluation of F on every iteration because the unlabeled EXIT would terminate the block in the THEN clause of the IF statement.

LOOPEXIT causes termination of iteration within the loop. If the labeled form is used, the label identifier must appear at the beginning of the loop statement.

Example D: Unlabeled LOOPEXIT.

```
FOR K = 0 STEP 1 UNTIL 23  DO
     IF A(K) EQL ACTIVE THEN
          BEGIN
          P = PX(K) ;
          AL = ALP(K) ;
          D = DE(K) ;
          LOOPEXIT ;
          END
     ELSE
          P = 0.0
     ENDIF
ENDDO
```

In this example, iteration terminates if a value for A(K) equals the value of ACTIVE.

Example E: Labeled LOOPEXIT.

```
L:  FOR J = 1 STEP 1 UNTIL K DO
          BEGIN
          K = N - M ;
          FOR I = J STEP-M UNTIL 1 DO
               IF INP(I+M) GEQ INP(I) THEN
                    LOOPEXIT L
               ELSE
                    INP(I+M) = W
               ENDIF
          ENDDO ;
          M = M / 2 ;
          END
     ENDDO
```

Here, the outer loop is terminated if conditions in the inner loop are met.

5.3.3 RETURN Statement

The form of a RETURN statement is:

```
RETURN [expression]
```

The effect of RETURN is to terminate the evaluation of a procedure body and transfer control to the point of call. A procedure body may contain more than one RETURN statement.

RETURN without an expression is used in procedures of type N or no type where no value is produced.

RETURN followed by a single expression is used with procedures of type H, I, D, R, and P. The possible types of the return expression are specified in Table 5-1. It is converted to the procedure type if necessary. See Section 6.5 for an example.

TABLE 5-1. RETURN EXPRESSION TYPE

Procedure Type	Return Expression Type
H	H, I, D, R
I	H, I, D, R
D	H, I, D, R
R	H, I, D, R
P	P

5.4 COMPILE UNITS

A compile unit is a non-empty sequence of declarations enclosed by BEGIN ... END FINIS brackets. The declarations are either data declarations or procedure declarations.

In general, a compile unit has the following structure:

```
CUNAME
     BEGIN
     d1 ;
     d2 ;
     .
     .
     .
     dn ;
     END FINIS
```

In the structure illustrated, CUNAME represents the compile unit name. This name is used by the compiler and by other THLL support tools to identify this compile unit and is therefore mandatory. The compile unit name is an identifier that should not contain any special character as one of the first eight characters. It is not followed by a semicolon.

The di represent declarations.

CHAPTER 6

PROCEDURE DECLARATIONS

A procedure declaration defines an identifier to be the name of a procedure. A procedure specifies code that can be executed from many places in a program by "invoking" or "calling" the procedure. A procedure call has the form:

```
P(al, ..., ak)
```

where P is the procedure name and al, ..., ak are the actual parameters. If the procedure has a value, then the call can be used in an expression position, otherwise the call must be used in a statement position.

The main part of a procedure is the procedure body which is a block that specifies the piece of code to be executed when the procedure is invoked. The procedure body is preceded by the procedure head which contains the following information:

A. Access of the procedure,

B. Type of the value of the procedure,

C. Name of the procedure,

D. List of formal parameters, and

E. Description of formal parameters.

The information expressed in items A, B, D, and E can be omitted. If omitted, certain defaults are assumed as explained below.

General format of a procedure declaration:

```
DEFINE
  access  type  PROCEDURE  procedure-name     /* head */
  formal-parameter-list ;
  formal-parameter-description ;

     BEGIN                                    /* body */
       .
       .
       .
     END
```

6.1 ACCESS PART

The access part describes how the procedure is used. It has significance only for the BP. It is needed here because of the architecture of the machine (case A below) and because the Monitor-Exec interface violates the THLL procedure interface conventions (cases C and D). There are four possible access specifications:

A. LINK

B. < empty > . This is the default if no access part is explicitly specified. For this case, it is assumed that the call and the procedure definitions are in the same virtual space.

C. EXEC

D. EXEC INTERRUPT n

LINK allows the call to the procedure and the procedure definition to be in different virtual spaces. Link procedures are not allowed to have entry points to procedures as arguments.

EXEC specifies that the procedure is treated by the Monitor as a Monitor called Exec Utility such as MODE INIT, EES, LINKED INTERRUPT, KEYBOARD PROCESSOR, etc. EXEC procedure declarations can appear only in privileged programs (see PRIV directive, Appendix C). An EXEC procedure saves the values of all registers on the Monitor's DATA.STACK on entry and restores the registers on exit from the procedure. An EXEC procedure cannot have arguments or a value. The entry point of an EXEC procedure cannot be passed to any procedure as a parameter.

EXEC INTERRUPT n means that the procedure is used as an interrupt procedure for level n. Therefore, n must be an integer in the range $0 \leq n \leq 47$. EXEC INTERRUPT n procedure declarations can appear only in privileged programs (see PRIV directive, Appendix C). An EXEC INTERRUPT n procedure first saves registers 0 and 1 on the Monitor's DATA.STACK, updates the Monitor CEIR with the currently executing interrupt routine, modifies the linkage registers to point to the executive, then turns the Monitor interrupt flag on and saves the rest of the registers and the old Monitor CEIR on DATA.STACK. The rest of the entry sequence is then the same as for the EXEC case after the registers have been saved. The exit sequence restores the original environment that existed when the interrupt procedure was entered. An interrupt procedure cannot have arguments or a value. The entry point of an EXEC INTERRUPT procedure cannot be passed to any procedure as a parameter.

6.2 TYPED PROCEDURES

Link procedures and regular procedures may or may not yield a value. If a value is defined, then the type of the value must precede the keyword PROCEDURE, otherwise no type specification precedes PROCEDURE. EXEC and EXEC INTERRUPT procedures cannot be valued. The possible types of a procedure are: HALF, INTEGER, DOUBLE, REAL, or POINTER.

The identifier following PROCEDURE is the name of the procedure. The procedure declaration binds this name to the procedure body.

6.3 FORMAL PARAMETERS

A procedure can have n arguments, n $\geq$ 0. These arguments can be referenced within the body symbolically by names. The special case of optional arguments are treated in Section 6.7. The names of all arguments are listed in the procedure head after the procedure name in the form of a list of identifiers separated by commas and enclosed in parentheses.

Example:

```
DEFINE PROCEDURE P(X1,X2,X3,X4) ;
...
```

This means that X1 is the name for the first argument of P, X2 the name for the second argument, etc. These argument names are called the formal parameters of the procedure.

When a procedure is invoked, the formal parameters are bound to the corresponding actual parameters. For each formal parameter, the binding mechanism and the type of the object this formal parameter is bound to must be specified. This is done in the description of the formal parameters. It follows the semicolon after the formal parameter list.

6.4 DESCRIPTION OF FORMAL PARAMETERS

The formal-parameters-description has two parts. They are:

A. Value part and

B. *Specification* part.

Each is described separately.

6.4.1 Value Part

This part describes the transmission mode of the actual parameters. THLL provides two binding mechanisms, also called argument transmission modes:

A. By value and

B. By reference.

If the formal parameter X is a value parameter, then it is treated as if it were a symbol declared in a fictitious block embracing the procedure body. That means X is local to the procedure and assignments made to X within the procedure have no effect outside of it. An actual parameter passed by value is unchanged after return from a procedure call.

If the formal parameter X is a reference parameter, then it is treated within the body of the procedure as if for every occurrence of X the actual parameter was substituted. That means that assignments made to X within the procedure have a permanent effect outside of it.

The value part appears immediately after the formal parameter list in the head of a procedure and is separated from it by a semicolon. All those parameters which are passed by value must be listed in the value part. It is of the form:

```
VALUE id1, id2, ..., idk ;
```

Each of the identifiers listed in the value part must be a formal parameter. Formal parameters not listed in the value part are, by default, reference parameters.

Rules for argument transmission:

A. Simple and subscripted variables of type H, I, D, R, P, and component variables representing full words may be passed by value or reference. Exception: on the BP, a subscripted array variable of type H can only be passed by value.

B. Array, stack, format, and device identifiers, simple variables and constants of type ALPHA and procedure entry points can be passed by reference only.

C. Constants (not of type ALPHA), component variables representing partial words or doublewords, and all expressions which are not variables can be passed by value only.

If no formal parameters are passed by value, then the value part, including the semicolon following it, is omitted in the procedure head.

6.4.2 Specification Part

The second part in the "description of the formal parameters" is the specification part. Every formal parameter must appear in the specification part. Those parameters which are to be transmitted by value appear in the value part and the specification part. If the procedure has no formal parameters, then the specification part, including the semicolon following it, is omitted.

In general, a formal parameter always represents a symbol of the following kind:

A. Simple variable,

B. Device,

C. Format,

D. Stack,

E. Array, or

F. Procedure.

For the most part, the specification of the formal parameters is quite simple. Only if the formal parameter represents a procedure can it become more complex.

The specification part follows the value part (if there is one) or the procedure head (if there is no value part).

The various kinds of parameters that can appear in the specification part are now discussed in detail.

6.4.2.1 Simple Variable, Device, or Format Specification - The form for specifying a formal parameter which is a simple variable is:

```
type id ;
```

where type is one of the keywords HALF, INTEGER, DOUBLE, REAL, POINTER, or ALPHA. Additional identifiers of the same type may appear if separated by commas.

The form for specifying a formal parameter which is a device or a format is:

```
DEVICE id ;                /* device specification */
FORMAT id ;                /* format specification */
```

Additional identifiers of the same kind may appear if separated by commas.

Example:

```
INTEGER FLAG,I,J ;
REAL X,Y,Z ;
DEVICE DISK ;
ALPHA MESS1 ;
FORMAT FM1,FM2 ;
```

Note that the size of the ALPHA variable is not given. An ALPHA variable is always passed by reference. When an ALPHA variable is used within a procedure, the size of the corresponding actual parameter is used.

6.4.2.2 Stack Specification - The form for specifying a formal parameter which is a stack is:

```
stack-type STACK id ;
```

where stack-type is one of the keywords HALF, INTEGER, DOUBLE, REAL, or POINTER. Additional stack identifiers of the same type may appear if separated by commas.

Example:

```
DOUBLE STACK A,B ;
POINTER STACK C ;
```

Note that the size of the stack is not given. This information is included in the stack header which is part of the stack data structure.

6.4.2.3 Array Specification - The form for specifying an array as a formal parameter is:

```
array-type ARRAY id1(size1), ..., id(size) ;
```

where array-type is one of the keywords HALF, INTEGER, DOUBLE, REAL, or POINTER; and id1, ..., id are identifiers specified as arrays. Each identifier is followed by a size specification in the form of one or two or three integer numbers, separated by commas and enclosed in parentheses or in the form of one or two or three stars.

If the size is of the form:

```
n1, n2, n3
```

then

```
0 ... n1 is the range of the first subscript,
0 ... n2 is the range of the second subscript, and
0 ... n3 is the range of the third subscript.
```

If the size of each dimension is specified by stars, then the array formal parameter has no specific size. For each invocation of the procedure, the size of the corresponding actual parameter becomes the size of the formal parameter. The size information is included in the header of the actual parameter. More efficient object code is produced if the ranges are specified by numbers.

If in an array specification a set of consecutively listed array names has the same number of dimensions and for each dimension the same range, then only the first array name in this collection needs a size specification. For the remaining names, the same size is assumed by default until a new size is specified.

Example:

```
INTEGER ARRAY DATA(*,*,*) ;
REAL ARRAY ALPH(15),DELTA,GAMMA ;
POINTER ARRAY PICK(7) ;
```

Note that the asterisks indicate that the integer array DATA has three subscripts. The real arrays ALPH, DELTA, and GAMMA all have 16 elements.

6.4.2.4 Procedure Specification - The form for specifying a formal parameter which is a procedure is:

```
proc-type PROCEDURE id ;
```

or

```
proc-type PROCEDURE id (arg-list) ;
```

where proc-type is one of the keywords HALF, INTEGER, DOUBLE, REAL, or POINTER if the procedure is typed, or it may be <empty> if the procedure is untyped. The first form is used for procedures without parameters, the second for procedures with parameters.

If the formal parameter procedure has parameters, arg-list must describe them in the order of occurrence in the procedure definition. This description contains for each parameter the information equivalent to the value part and the specification part in the procedure definition.

The syntax for arg-list is as follows:

arg-list is a sequence of specification elements, separated by commas. Each specification element corresponds to a formal parameter in the order of the occurrence in the procedure definition. A specification element can have one of the following forms:

```
type                          /* simple variable by reference */
VALUE type                    /* simple variable by value */
type VALUE                    /* simple variable by value */
type STACK                    /* stack */
FORMAT                        /* format */
ALPHA                         /* ALPHA variable */
DEVICE                        /* device */
type ARRAY (size)             /* array */
type PROCEDURE (arg-list)     /* valued procedure */
PROCEDURE (arg-list)          /* unvalued procedure */
type PROCEDURE                /* valued procedure without parameters */
PROCEDURE                     /* unvalued procedure without parameters */
```

where type is one of the keywords HALF, INTEGER, DOUBLE, REAL, or POINTER, and size has the same meaning as in the array specification. Thus, size consists of one or two or three integer numbers, or of one or two or three asterisks. The last two lines describe the cases where the procedure being passed as a parameter has a parameter which is itself a procedure.

The following examples illustrate specification elements, that can occur in a specification part, describing procedure formal parameters.

Examples:

```
REAL PROCEDURE CHECK ;    /* 1 */

PROCEDURE STALL ;         /* 2 */

INTEGER PROCEDURE COUNT(POINTER VALUE,REAL ARRAY(9),INTEGER STACK,
 INTEGER VALUE) ;         /* 3 */

PROCEDURE ZIP(REAL,REAL PROCEDURE(INTEGER,VALUE POINTER),DOUBLE) ;
                          /* 4 */
```

Example 1 illustrates a typed procedure without parameters.

Example 2 describes an untyped procedure without parameters.

Example 3 describes a procedure of type INTEGER which has four formal parameters. Two of these parameters are to be passed by value. Since arrays and stacks must be passed by reference, only their attributes are given. A size is required for the array, but not for the stack.

Example 4 describes an untyped procedure ZIP which requires a procedure as the second parameter. Therefore, the argument list for that procedure parameter must be given. This list in turn consists of two parameters, the first one being an integer variable by reference, the second one a pointer variable by value.

6.5 PROCEDURE HEAD EXAMPLE

This example combines all parts of the procedure head. The comments within the example indicate the uses of the various parameters. Since the procedure PI is of type integer, note that the return statement contains a call to the integer procedure Z. This shows the use of the correct RETURN statement form (see Section 5.3.3).

The call to procedure PI could appear in a statement, but it must have three actual parameters. Such a call might be:

```
IF PI(5-X,RMATRIX,ENTRYP IPROC) THEN
     V =  SIN(M)
ELSE
     V = -SIN(M)
ENDIF ;
```

This assumes that declarations have been made for X, RMATRIX, IPROC, V, and M. Such as:

```
INTEGER X ;
REAL ARRAY RMATRIX(3,7) ;
EXTERNAL INTEGER PROCEDURE IPROC(REAL,POINTER) ;
REAL V,M ;
```

Example:

```
DEFINE INTEGER PROCEDURE PI(I,Y,Z) ;
  VALUE I ;                /* value part */
          /* specification part follows: */
  INTEGER I ;              /* INDEX FOR GLOBAL ARRAY WTABLE */
  REAL ARRAY Y(*,*) ;      /* PRECEDENCE MATRIX */
  INTEGER PROCEDURE Z(REAL,POINTER) ;
                           /* THE VALUE OF Z WILL BE RETURNED AS
                              THE VALUE OF PI.  THE FIRST ARGUMENT
                              OF Z IS AN ELEMENT OF Y, THE SECOND
                              ARGUMENT IS A GLOBAL POINTER
                              VARIABLE Q */
          /* end of specification part */
```

```
BEGIN
  .
  .
  .
RETURN Z(Y(1,2),Q) ;
END
```

Thus, the specification part gives, for each formal parameter, information about the type and the kind of entity it represents. If the formal parameter represents an array, then each dimension may be specified by a number or by an asterisk, in which case no fixed ranges for the subscripts are assumed. If the formal parameter represents a procedure, then its type must be specified as well as each argument that the procedure takes. Since any of those arguments can again be a procedure entry point, it is clear that the specification part for a formal parameter representing a procedure can have a nested structure to any depth. Therefore, the definition of the construct arg-list given above is recursive; it contains the construct arg-list on the right side of the definition.

It is good programming practice to use a standard format for procedure declarations similar to the one in the example above. In particular, for each formal parameter a brief explanation should be given telling what it means or how it is used.

It is possible for a procedure to have no arguments. In that case, the formal parameter list and the description of the formal parameters are omitted. That means that the procedure body follows directly after the procedure name, separated from it by a semicolon.

6.6 GENERAL PROPERTIES OF PROCEDURES

Formal parameters are not permitted in interrupt procedures. Upon entry, the Interrupt procedure saves all the active general registers on an executive stack, and then uses the runtime stack like any other THLL procedure.

All procedures can be called in a recursive manner. A procedure may call itself or it can call a procedure which eventually causes it to be called. Upon each entry, the procedure saves the present environment on a stack. Actual parameters, temporary storage, and shared variables are allocated for each invocation of a recursive procedure. Therefore, the contents of actual parameters, temporary storage and shared variables will not be destroyed by another invocation of the procedure. However, OWN data will be destroyed.

The following description is unique for the BP:

> Data from all virtual spaces are available to procedures in each virtual space. Executable instructions in "brother" nodes are overlaid in virtual space. A procedure in a "brother" virtual space must be invoked via the link instruction (see external declaration);

but, since the data are not included in the virtual overlay, the data are available to the "brother" virtual space. The Binder must process procedures of this type to assure that the data are moved to a common virtual space (Reference 6).

When a procedure is called, some preparation is made by the calling procedure. Automatic type conversion of the actual parameter to the type of the corresponding formal parameter is performed for parameters that are passed by value. An error message is emitted when the types do not agree for parameters passed by reference, and no type conversion is performed.

The following table shows the legal type conversions from actual parameter type to formal parameter type for the corresponding argument that is passed by value.

		actual parameters					
		H	I	D	R	P	A
	H	H	H	H	H	*	*
	I	I	I	I	I	*	*
formal	D	D	D	D	D	*	*
parameters	R	R	R	R	R	*	*
	P	*	*	*	*	P	*
	A	*	*	*	*	A	A

This type table is almost identical to the type table used for the assignment operator (=). If a formal parameter is of type A, then an actual parameter of type P is assumed to point to an ALPHA variable. If the formal parameter is of type P, then the corresponding actual parameter may be the integer 0. If the formal parameter is of type D, then the corresponding actual parameter may be a string constant. Only the first eight characters of the string constant are moved to the double. The string is considered left-justified and blank-filled.

If the formal parameter is a format, then the actual parameter can be format, pointer or integer constant zero. The pointer is assumed to point directly to a format. The integer constant zero means that no format is being used.

If the number of actual parameters is less than the number of formal parameters, TRICOMP adds the following actual parameters for the missing arguments.

A. 0 of the appropriate type for a formal parameter passed by value.

B. ENTRYP of the called procedure for a parameter passed by reference.

On the BP and VAX, a runtime interrupt occurs if the procedure tries to reference a nonexistent parameter passed by reference.

6.7 OPTIONAL ARGUMENTS

A procedure P with n $\geq$ 0 formal parameters may have optional arguments. This is indicated in the procedure declaration by writing the keyword OPTARG after the last formal parameter, e.g.,

```
DEFINE INTEGER PROCEDURE P(X1, ..., Xn, OPTARG) ;
```

or

```
DEFINE INTEGER PROCEDURE Q(OPTARG) ;
```

and similarly for external procedure declarations.

This means that in addition to the first n $\geq$ 0 parameters which can be referenced within the body of P directly by name, any remaining actual parameters an+1, ..., am of a call:

```
P(a1, ..., an, an+1, ..., am)
```

can be referenced within the body of P indirectly using the special standard functions described below.

A. ARGCNT - is an integer procedure with no arguments. Its value is the number of actual parameters of the current procedure.

B. ARGPTR(I) - is a pointer procedure of one argument; I > 0 is an integer. The value of ARGPTR(I) is a pointer to the I'th actual parameter or to a memory word containing the value of the actual parameter. The value of ARGPTR(I) is a pointer to a copy of the actual parameter for the following cases:

Partial word component variable,
subscripted array variable of type half (for BP only),
entry point of a procedure,
expression that is not a variable.

C. ARGTYPE(I) - is an integer procedure of one argument; I > 0 is an integer. The value of ARGTYPE is an integer representing the type of the I'th argument of the current procedure. The possible values are tabled below:

Argument Type	Value Returned
N (no type)	0
H	1
I	2
D	3
R	4
P	5
A	6

D. ARGSYNCL(I) - is an integer procedure of one argument; I > 0 is an integer. The value of ARGSYNCL is an integer representing the "syntactic class" of the I'th argument as follows. The meaning of the value of ARGSYNCL is machine-dependent:

Argument Class	Value Returned	
	BP	VAX,MC68000
one-dimensional array	1	1
two-dimensional array	2	2
three-dimensional array	3	3
simple variable	0	4
stack	4	5
procedure	0	6
device	0	7
format	0	8

Optional arguments are useful for defining procedures with a variable number of arguments. Thus, the user can do what is presently being done on the system level for built-in procedures like READ or WRITE.

If the procedure P has n formal parameters and P is not declared as a procedure with optional arguments, then in a call to P, all actual parameters are evaluated, but only the first n parameters are bound to their corresponding formal parameters. A call to ARGCNT within P would yield the value n.

On the VAX and on the MC68000, calls to ARGTYPE and ARGSYNCL should only be used in a procedure declared with optional arguments. The external declaration for a procedure with optional arguments must specfify OPTARG.

6.8 RETURNING A VALUE OF A PROCEDURE

As mentioned above, a procedure may be typed or untyped. A typed procedure returns a value which has the type as specified in the head. The possible types are: H, I, D, R, and P. An untyped procedure does not return a value. It is executed for its effect on its environment only. Typed procedures are called functions. For convenience, the improper type N (no type) is assigned to an untyped procedure, which is also called a function. Interrupt and executive procedures must be of type N.

Return from a procedure is made via a return statement. There are two kinds of return statements:

A. Unvalued return statement: RETURN

B. Valued return statement: RETURN < return expression >

Any number of return statements can appear within a procedure body. If the procedure is untyped, then only unvalued returns should occur. If the procedure is typed, then only valued returns should appear. All return expressions should have types compatible with the type of the procedure according to the type rules for the assignment operator (=).

Procedure Type	Type of Return Expression
H	H I D R
I	H I D R
D	H I D R
R	H I D R
P	P

At the end of a procedure, a default RETURN is implied:

RETURN, for an untyped procedure
RETURN EO, for a typed procedure

Procedure Type	EO
H	0
I	0
D	0
R	0.0
P	0

CHAPTER 7

GLOBAL, EXTERNAL, AND COMMON DECLARATIONS

7.1 GLOBAL AND EXTERNAL DECLARATIONS

For program development, it is advantageous to write and check out small compile units. Integration into larger units can be done by combining the components into a new THLL program and compiling it or by "binding" the precompiled units together (Reference 6). The latter approach does not require recompilation and allows the formation of virtual and physical overlay structures. Therefore, the binding approach is recommended for program integration.

How do compile units communicate with each other? THLL provides a mechanism for equivalencing names in separately compiled units. This is done by GLOBAL and EXTERNAL declarations.

The GLOBAL declaration makes an identifier, defined in one compile unit, known to a separate compile unit. An EXTERNAL declaration defines an identifier to represent an entity which is defined under the same name in another separate compile unit.

7.1.1 Global Declaration

A GLOBAL declaration consists of the keyword GLOBAL followed by a sequence of identifiers separated by commas. The effect is that each identifier in the list is made available to the Linking Loader to satisfy external names in other compile units. These identifiers must be properly declared within the same block of the compilation in which they are declared GLOBAL. Components, devices, labels, and switches cannot be declared GLOBAL.

Example:

```
GLOBAL  A,SQUARE,SCAN,COMPARE ;
```

Note that no attributes are given in the GLOBAL declaration. From the declaration alone, a reader cannot separate array, procedure, simple variables, etc. from one another.

Global procedures must be declared in the outermost block of a compile unit.

7.1.2 External Declaration

An EXTERNAL declaration does not define an identifier, it just equivalences it to an identifier defined outside of this compile unit. However, sufficient information about the identifier, its type, its dimensions if it is an array, its arguments if it is a procedure, must be given to allow the identifier declared EXTERNAL to be treated properly. This complicates the EXTERNAL declaration slightly as compared to the GLOBAL declaration.

Simple variables, formats, arrays, stacks, and procedures can be declared external. Each is described in the following sections. Components, devices, labels, and switches cannot be declared external.

7.1.2.1 External Simple Variable and Format Declaration - The form of an external declaration for simple variables is:

```
EXTERNAL    type  id1, id2, ..., idk
```

where type is one of the type keywords HALF, INTEGER, DOUBLE, REAL, POINTER, or ALPHA. Each identifier in the list following the type keyword is declared as an external variable of the indicated type.

The form of an external declaration for format identifiers is:

```
EXTERNAL    FORMAT id1, id2, ..., idk
```

Each identifier in the list following the keyword FORMAT is declared as the name of an external format.

Examples:

```
EXTERNAL POINTER P,HERE ;
EXTERNAL ALPHA MESSAGE ;
EXTERNAL FORMAT PRERR ;
```

Note that no length information is required for ALPHA declarations.

7.1.2.2 External Stack Declaration - The form for an external declaration of a stack is:

```
EXTERNAL type STACK id
```

where type is one of the keywords HALF, INTEGER, DOUBLE, REAL, or POINTER. Additional identifiers of the same type may appear if separated by commas.

Examples:

```
EXTERNAL DOUBLE STACK DS ;
EXTERNAL POINTER STACK TP,FP,AP ;
```

Note that no size information is required.

7.1.2.3 External Array Declaration - An external array declaration has the following form:

```
EXTERNAL  type  ARRAY idl(sizel), ..., id(size)
```

where type is one of the keywords HALF, INTEGER, DOUBLE, REAL, or POINTER. idl, ..., id are identifiers declared as external arrays of the indicated type. Each identifier is followed by a size specification in the form of one, two, or three integer numbers separated by commas and enclosed in parentheses. The size can also take the form of one, two, or three asterisks separated by commas and enclosed in parentheses. Obviously, this specifies only the number of dimensions but not the number of elements in the external array. If in an external array declaration a set of consecutively listed array names has the same number of dimensions and for each dimension the same range, then only the first array name in this collection needs a size specification. For the remaining names, the same size is assumed by default until a new size is specified.

If the number of dimensions and the upper bound of each subscript are included in an external array declaration, this information is used by the compiler to perform certain compile time checks to ensure proper array subscripting. If the upper bound of each array subscript is indicated by *, then the upper bound is variable and can be accessed by looking at header information which is stored along with the array storage block. In this case, bounds checking must always be done at execution time and may result in less efficient code.

Examples:

```
A.  EXTERNAL REAL ARRAY T(2) ;

B.  EXTERNAL POINTER ARRAY X(*,*),Y(1) ;
```

7.1.2.4 External Procedure Declaration - An external procedure declaration has one of the following forms:

```
EXTERNAL type lnk PROCEDURE id (arg-list)
```

or

```
EXTERNAL type lnk PROCEDURE id
```

where type is one of the keywords HALF, INTEGER, DOUBLE, REAL, or POINTER if the procedure is valued, or it is <empty> if the procedure is unvalued. lnk is either the keyword LINK or is <empty>. The procedure name id is followed by arg-list, the list of argument descriptions. The syntax for arg-list is described in the section on procedure specification (Section 6.4.2.4).

If the external procedure requires arguments, they must be described in the order of occurrence within parentheses. It is also necessary to indicate the transmission mode of these arguments.

If the formal parameter is an array, then the upper bound of each subscript may be indicated the same as for external arrays.

Examples:

```
A. EXTERNAL INTEGER PROCEDURE A(REAL,INTEGER ARRAY(*,*)) ;

B. EXTERNAL PROCEDURE Y(REAL ARRAY(5,10),POINTER PROCEDURE(INTEGER)) ;

C. EXTERNAL LINK PROCEDURE Z(REAL VALUE,POINTER STACK) ;
```

Note that in Example C the first parameter of the LINK procedure Z is passed by value rather than reference. Unless specified to be VALUE, all parameters are assumed to be passed by reference.

7.1.3 Effect and Use of Global and External

In order to describe the effect and use of GLOBAL and EXTERNAL declarations more precisely, we have to consider an entire program which is the collection of independently compiled compile units combined together into an object program structure consisting of one or more physical overlays. The individual compile units of a program may all be in different virtual spaces or some of them may be in the same virtual space. Likewise, they may be in different physical overlays or they may all be in the same physical overlay.

In general, GLOBAL identifiers must be unique within the program. For BP programs, more flexibility is provided (see Reference 6). Within a compile unit, an identifier can be declared GLOBAL or EXTERNAL, not both. The compile unit containing the GLOBAL declaration also contains the definition of the identifier causing storage allocation for variable, array, or stack ids, data generation for format ids, and code generation for procedure ids. The EXTERNAL declaration appearing in a compile unit only equivalences the identifier with a symbol defined outside of this compile unit. It follows that for each identifier declared external in one or more of the compile units in the program there must be exactly one GLOBAL declaration in a different compile unit of the program. GLOBAL definitions and external references cannot be in parallel physical overlays.

7.1.4 Scope of Declarations

The scope of identifiers within a compile unit is strictly determined by the block structure of the compile unit. This is even true for identifiers declared GLOBAL or EXTERNAL. If X is declared GLOBAL in the compile unit M then X can be referenced in M only within its scope in M. If X is declared GLOBAL in M and EXTERNAL in separately compiled compile units Mi, ..., Mk, then X can be referenced in each Mi within its scope in Mi. The effect of GLOBAL X and EXTERNAL X is that references to X in M and in each Mi actually refer to the same entity--the same storage location for data and format identifiers, the same entry point for a procedure identifier. Therefore, the effective "scope of X in the program" is the union of the individual scopes of X in M and in each M1, M2, ..., Mk.

All GLOBAL variable, array, and stack identifiers must have the allocation mode OWN. GLOBAL procedures must be defined in the outermost block.

7.1.5 Procedures and Link Procedures

A procedure may be declared as PROCEDURE or LINK PROCEDURE in the EXTERNAL declarations. The corresponding EXTERNAL declaration should also say PROCEDURE or LINK PROCEDURE. The distinction between regular procedures and LINK procedures is significant only on the BP.

A LINK PROCEDURE is invoked using a LINK assembly instruction while those defined as a PROCEDURE are invoked using a call instruction. The call instruction can be used to invoke procedures in the same virtual space (in the same node or a "father" node in a root segment structure). A call instruction can be used to invoke procedures in a "son" node if the user knows that the "son" node is in the same virtual space. The LINK instruction reloads base registers in the BP, and it should be avoided when possible. It is suggested that a minimum number of virtual spaces be used to span the program. The programmer is given the responsibility and versatility of controlling linkages between nodes.

Procedure entry points may not be passed to a LINK procedure. An entry point for a LINK procedure, EXEC procedure, or EXEC INTERRUPT procedure cannot be passed to any procedure.

7.2 COMMON DECLARATION

The COMMON declaration provides a mechanism for defining a block of memory for OWN data such that only the origin of this block is a GLOBAL symbol and that various parts of this data block can be referenced symbolically by names. It allows therefore, in effect, the free use of symbols for communication between compile units without overloading the capacity of the LINKER for handling EXTERNALS.

The format for the COMMON declaration is:

```
ge  COMMON  org ;
    D1 ;
    .
    .
    .
    Dk ;
    ENDCOM ;
```

where

ge is either GLOBAL or EXTERNAL,

org is an identifier,

and Di is a declaration for simple variables, for arrays, or for stacks.

If ge is the keyword GLOBAL, then the COMMON declaration is called a global common; and the symbol org is in this case a GLOBAL symbol. Otherwise, the declaration is called an external common and org is in this case an EXTERNAL symbol. The rules for GLOBALS/EXTERNALS apply, which means, among other things, that the symbol org must be unique in the set of compile units using this COMMON declaration.

A global common causes OWN memory to be allocated. The memory allocated for all symbols defined within a global common is called a common block. An external common does not cause allocation of memory. It just informs the compiler about the relative position of the various common symbols within the common block. The symbol org serves only to define an origin for a common block, it cannot be used by the programmer directly in the program. For debugging purposes, however, the programmer can use it. To TOADC the global common origin looks exactly like a global integer one-dimensional array.

If a number of compile units reference symbols defined within a common declaration, then one compile unit should contain the global common and all others should contain the external common for the same origin and for exactly the same sequence of declaration D1, ..., Dk. Actually, the name of a data element defined in a certain position within a common could be different in different compile units; it is the position that counts. However, using different names is not a good programming practice.

The common block defined in a global common declaration may be longer than the common block specified in an external common declaration with the same org identifier.

All symbols defined within a common are considered OWN. However, for reasons given at the end of this section, OWN must not be specified explicitly.

Example:

```
   GLOBAL COMMON ORG1 ;
        INTEGER LAT,LONG,RANGE ;
        REAL ARRAY DUMP(10) ;
        INTEGER NEW ;
        DOUBLE STACK STCK(5) ;
        POINTER PTR ;
   ENDCOM ;
```

On the BP, the common block for this declaration is laid out in memory as follows:

ORG Identifier	Word in Memory	Data Description
ORG1	0	LAT
	1	LONG
	2	RANGE
	3	hole
	4	header for DUMP
	5	
	6	DUMP(0)
	7	
	8	DUMP(1)
	...	...
	26	DUMP(10)
	28	NEW
	29	hole
	30	header for STCK
	31	
	32	12 words for STCK
	...	...
	44	PTR

All symbols defined within a global (external) common are not considered global (external) by the compiler and they are therefore not passed on to the assembler as globals (externals). This also means that symbols within a common can be redefined in other blocks of the same compile unit for different purposes. Such symbols are not considered multiply-defined by the compiler.

It is recommended that insert files be used for common declarations excluding the first line which defines the origin as GLOBAL or EXTERNAL.

Example:

```
   GLOBAL COMMON ORG1 ;
   INSERT INSORG1(FN) ;
```

should appear in the compile unit that is to contain the global common while:

```
EXTERNAL COMMON ORG1 ;
INSERT INSORG1(FN) ;
```

should appear in all compile units that use the same common as an external common.

The member INSORG1 on the file FN looks like this:

```
/* COMMON ORG1 */
D1 ;
.
.
.
Dk ;
ENDCOM ; /* END OF COMMON ORG1 */
```

This technique allows programmers to maintain only one single copy for each common on an insert file. The same individual declaration Di appears then within a global common and within an external common. Since EXTERNAL OWN makes no sense and is treated as an error by the compiler, therefore, the symbols defined within a common should not be declared OWN.

Within a compile unit, the scope of a symbol declared within a COMMON declaration is the same as the scope of the COMMON declaration.

CHAPTER 8

INPUT/OUTPUT

This chapter describes the facilities for formatted Input/Output (I/O) provided in THLL. Transferring data to or from an external device requires the specification of a device, a format describing the conversion of the data, and the actual data. Device and format identifiers are given their meaning via declarations. The actual data to be transferred are specified as a sequence of actual parameters in the READ or WRITE call. This sequence of parameters is called the I/O list. The constructs that can appear in a I/O list and their processing in conjunction with a format are described in the section on Format Processing.

8.1 DEVICE DECLARATION

A device declaration defines an identifier to be the name of a TDCC hardware device. A device identifier can be used as an actual parameter in a procedure call. The only time it has to be used is in a call to the procedure OPEN which opens a file for the specified device.

The device declaration has the following form:

```
DEVICE id = dev
```

where id is an identifier and dev is one of the device key words listed in Table 8-1.

For convenience, a set of device declarations:

```
DEVICE id1 = dev1 ;
   .
   .
   .
DEVICE idk = devk ;
```

can equivalently be written as:

```
DEVICE id1 = dev1, ..., idk = devk ;
```

TABLE 8-1. DEVICE NAMES

Device Name	Peripheral Device
SPRINT	system printer (console)
CPRINT	computer printer (DCC)
MDF	magnetic disk file
MTF	magnetic tape file
KBDSS	keyboard display subsystem
ICL	inter-computer link

Example:

```
DEVICE DISK = MDF, TAPE = MTF, KB = KBDSS ;
```

Here, three device identifiers are defined as names for the MDF, the MTF, and the KBDSS, respectively.

8.2 FORMAT DECLARATION

A format declaration defines an identifier to be the name of a format. A format is a list of items that are interpreted during an I/O operation for the purpose of converting data from their internal representation to their external representation as character strings and vice versa.

A format declaration has the following form:

```
FORMAT id (format-list)
```

where id is an identifier and format-list is a sequence of format items separated by commas.

A format item has one of the following forms:

```
R'rn'
[$] S 'n'
[$] str
[$] I '[-]n'
[$] D '[-]n'
[$] O '[-]n'
[$] H '[-]n'
[$] [P'm'] F '[-]n'
[$] E 'n'
[$] L 'n'
[$] A 'n'
k (format-list)
```

where rn, n, m, and k are unsigned integer numbers; and str is a THLL string.

When a format is specified for a READ or WRITE operation, the format items that appear in the format list are used from left to right in conjunction with an I/O list containing variables and values. The I/O list may also contain loop arguments which are expanded into a list of variables and values at execution time.

The first three format items above do not require to be matched with an element from the I/O list, they have an effect by themselves. All other format items, except the last one, require to be matched up with an element from the I/O list for which the format item has the defined effect. Processing format items in conjunction with an I/O list are described in Section 8.2.4.

The format item of the form k(format-list) is a short form for repeating the format-list in parentheses k times.

With one exception, the integer number n in the above format items specifies the field size, that is the number of characters read on input or the number of characters produced on output. If the field size is preceded by a minus sign, then the entire field must be filled with binary, octal, decimal, or hexadecimal digits depending on the type of the number under consideration. If the number does not fill the entire field, then leading zeros are produced on output and are expected on input. If the field size of a format item is preceded by a minus sign, then negative values produce a sequence of *'s on output and signed numbers are illegal on input.

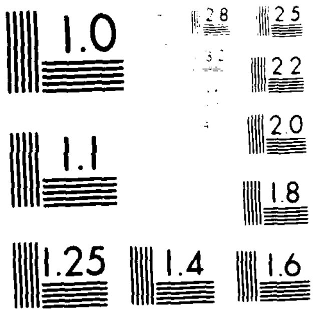
The exception is an I'n' or [P'm']F'n' format item preceded by a string format item ending in the special character $\pm$. In this case, the size of the field of the I'n' or [P'm']F'n' number is n+1; and the size of the string is one less than the size of the string length. The first position in the n+1 characters in the numeric field is set to the sign of the value on output and must be a sign on input. Both adjacent format items must have the same memory protection, that is, either both have $ specified or neither has.

The $ sign is optional for most format items. It is significant only for a WRITE operation to one device, the KBDSS. If a format item includes a $ sign, then on output of the character sequence for that format item each character in this sequence has the memory non-protect bit set. The corresponding field on the display of the KBDSS can then be changed by the KBDSS operator. A field on the KBDSS that corresponds to a format item not including a $ sign is protected and cannot be changed by the KBDSS operator.

AD-A145 116 THLL (TRIDENT HIGHER LEVEL LANGUAGE) REFERENCE MANUAL (U) NAVAL SURFACE WEAPONS CENTER DAHLGREN VA H G HUBER JUL 84 NSWC/TR-84-101

2/3

UNCLASSIFIED F/G 9/2 NL


1.0
2.8
2.5
3.2
2.2
3.6
4.0
2.0
1.1
1.8
1.25
1.4
1.6

```
[$] H '[-]n'
[$] [P'm'] F '[-]n'
[$] E 'n'
[$] L 'n'
[$] A 'n'
k (format-list)
```

where rn, n, m, and k are unsigned integer numbers; and str is a THLL string.

When a format is specified for a READ or WRITE operation, the format items that appear in the format list are used from left to right in conjunction with an I/O list containing variables and values. The I/O list may also contain loop arguments which are expanded into a list of variables and values at execution time.

The first three format items above do not require to be matched with an element from the I/O list, they have an effect by themselves. All other format items, except the last one, require to be matched up with an element from the I/O list for which the format item has the defined effect. Processing format items in conjunction with an I/O list are described in Section 8.2.4.

The format item of the form k(format-list) is a short form for repeating the format-list in parentheses k times.

With one exception, the integer number n in the above format items specifies the field size, that is the number of characters read on input or the number of characters produced on output. If the field size is preceded by a minus sign, then the entire field must be filled with binary, octal, decimal, or hexadecimal digits depending on the type of the number under consideration. If the number does not fill the entire field, then leading zeros are produced on output and are expected on input. If the field size of a format item is preceded by a minus sign, then negative values produce a sequence of *'s on output and signed numbers are illegal on input.

The exception is an I'n' or [P'm']F'n' format item preceded by a string format item ending in the special character $\pm$. In this case, the size of the field of the I'n' or [P'm']F'n' number is n+1; and the size of the string is one less than the size of the string length. The first position in the n+1 characters in the numeric field is set to the sign of the value on output and must be a sign on input. Both adjacent format items must have the same memory protection, that is, either both have $ specified or neither has.

The $ sign is optional for most format items. It is significant only for a WRITE operation to one device, the KBDSS. If a format item includes a $ sign, then on output of the character sequence for that format item each character in this sequence has the memory non-protect bit set. The corresponding field on the display of the KBDSS can then be changed by the KBDSS operator. A field on the KBDSS that corresponds to a format item not including a $ sign is protected and cannot be changed by the KBDSS operator.

8.2.1 Semantics of Format Items on Input

The following table lists for each format item its meaning in terms of the action on a substring of the character sequence representing the input. N is the length of the substring, N and M are integers.

Format Item	Action
R'N'	Skip over next N carriage returns.
S'N'	Skip over next N characters.
str	Skip over next LENGTH(str) characters.
I'N'	Generate integer value from a decimal string.
D'N'	Generate integer value from a binary string.
O'N'	Generate integer value from an octal string.
H'N'	Generate integer value from a hexadecimal string.
[P'M']F'N'	Generate real value from a floating point string with possible exponent ignoring P'M'.
E'N'	Generate real value from floating point string with possible exponent.
A'N'	Use input string as is.
#/±/,I'-N'	Generate integer value from decimal string assumed to be left-filled with zeros and preceded by a sign.
#/±/,[P'M']F'-N'	Generate real value from floating point string assumed to be left-filled with zeros and preceded by a sign.

8.2.2 Semantics of Format Items on Output

The following table lists for each format item its meaning in terms of the conversion of a value to a string of characters which is part of the output. For the first three format items, a string of characters is directly produced; no value is required to be converted. For all other format items, a value is matched up with the format item and converted as described.

When a positive N is specified then the field size is N. Both N and M are integers.

Format Item	Action
R'N'	Generate N carriage returns; for N=0, generate form feed.
S'N'	Generate N spaces.
str	Generate the string.
I'N'	Generate a decimal integer character string.
D'N'	Generate a binary integer character string.
O'N'	Generate an octal integer character string.
H'N'	Generate a hexadecimal integer character string.
[P'M']F'N'	Generate a floating point number character string with a decimal fraction of M places. If P'M' is omitted then P'0' is assumed.
E'N'	Generate a floating point number character string with an exponent.
L'N'	Generate a T if the Boolean is true or F if the Boolean is false, left-justified in the field.
A'N'	If N = 0, the field size equals the number of characters in the ALPHA variable. HALF, INTEGER, DOUBLE, and REAL variables may be output under A format. Each is be broken into 8-bit fields for interpretation as characters. Thus, each type has

	an implied number of characters to be generated. These implied numbers are: HALF is 2; INTEGER is 4; DOUBLE and REAL are 3. A pointer to an ALPHA variable ›roduces the same result as the ALPH\ variable name.
#/±/,I'-N'	Generate a decimal nteger character string left-filled with zeros and preceded by the sign of the value.
#/±/,[P'M']F'-N'	Generate a floating point number string left-filled with zeros and preceded by the sign of the value.

When variables are converted to E or F format, rounding occurs. If the converted value does not fit into the field specified by the corresponding format item, the field is filled with asterisks. If the converted value does not fill the field, it is right-justified and filled with blanks or with leading zeros if the optional minus sign appears with types I, O, H, F, or D. For ALPHA format, if N is less than the actual or implied number of characters, the converted variable is filled with asterisks. If N is greater than the actual or implied number of characters, the field is left-justified and blank-filled.

8.2.3 Format Examples

8.2.3.1 WRITE Printer/Keyboard Formats -

Format Item	Digits	Spaces	Sign	Point
D'N'	0-1	Yes	No	No
D'-N'	0-1	No	No	No
O'N'	0-7	Yes	No	No
O'-N'	0-7	No	No	No
H'N'	0-F	Yes	No	No
H'-N'	0-F	No	No	No
I'N'	0-9	Yes	Minus	No
I'-N'	0-9	No	No	No
#/±/, I'-N'	0-9	No	Yes	No
F'N'	0-9	Yes	Minus	Yes
F'-N'	0-9	No	No	Yes
#/±/, F'-N'	0-9	No	Yes	Yes
E'N'	0-9	No	Yes	Yes
L'N'	T,F	Yes	No	No
A'N'	64-Character set			

NOTES: N is an integer; M is an integer.

1. A value of length greater than N or a negative value with I'-N' or F'-N' writes all asterisks in the field.

2. If P'M' is not specified with F'N' or F'-N', then P'0' is assumed.

3. F'N' or F'-N' uses two digits to left of point; i.e., -0. or 10. for N = 3.

4. E'9' is smallest E specification; i.e., ±.XE±XXXX.

5. Types D, O, and H are considered bit patterns, not signed numbers.

8.2.3.2 READ Keyboard Formats -

Format Item	Digits	Spaces	Sign	Point
D'N'	0-1	No	No	No
D'-N'	0-1	No	No	No
O'N'	0-7	No	No	No
O'-N'	0-7	No	No	No
H'N'	0-9	No	No	No
H'-N'	0-9	No	No	No
I'N'	0-9	No	Optional	No
I'-N'	0-9	No	No	No
#/±/, I'-N'	0-9	No	Required	No
F'N'	0-9	No	Optional	Yes
F'-N'	0-9	No	No	Yes
#/±/, F'-N'	0-9	No	Required	Yes
'N'	0-9	No	Optional	Yes
A'N'	0-9	Yes	Yes	Yes

NOTES: N is an integer; M is an integer.

1. If P'M' is not specified with F'N' or F'-N', then P'0' is assumed.

2. A numeric field that contains <u>all</u> spaces or <u>all</u> underlines is not processed; i.e., the corresponding list element is <u>not</u> altered.

3. There is no T or F on the keyboard for L'N' input.

8.2.3.3 WRITE Printer/Keyboard Format Examples -

Format Item	Value = 0	Value = 1	Value = -1	
D'2'	b0	b1	**	Note 2, 5
D'-2'	00	01	**	Note 2, 5
O'2'	b0	b1	**	Note 2, 5
O'-2'	00	01	**	Note 2, 5
H'2'	b0	b1	**	Note 2, 5
H'-2'	00	01	**	Note 2, 5
I'2'	b0	b1	-1	
I'-2'	00	01	**	Note 3
#/±/, I'-2'	+00	+01	-01	
F'3'	b0.	b1.	-1.	
F'-3'	00.	01.	***	Note 3
#/±/, F'-3'	+00.	+01.	-01.	
E'9'	+.0E+0000	+.1E+0001	-.1E+0001	
E'8'	********	********	********	Note 4
L'2'	Fb	Tb	Tb	

NOTES:

1. b means space.
2. Field too short for value; i.e., the integer -1 = FFFFFFFF(HEX).
3. Sign illegal in I'-2' and F'-3' format.
4. Field too short for value.
5. Types D, O, and H are considered bit patterns, not signed numbers.

8.2.3.4 READ Keyboard Format Examples -

Format Item	Legal	Illegal
D'2'	01	b1, +1, -1, 02
D'-2'	01	b1, +1, -1, 02
O'2'	01	b1, +1, -1, 08
O'-2'	01	b1, +1, -1, 08
H'2'	01	b1, +1, -1
H'-2'	01	b1, +1, -1
I'2'	01, +1, -1	b1
I'-2'	01	b1, +1, -1
#/±/, I'-2'	+01, -01	b01, +b1
F'3'	01., +1., -1., 001	b1.
F'-3'	01., 001	b1., +1., -1., bb1

#/$\underline{+}$/, F'-3'	+01., -01., +001	b01., +b1, +bb1
E'$\overline{9}$'	+.1E+0001, -.1E+0001, 0.1E+0001	b.1E+0001

NOTES:

1. b means space.

2. The decimal point in the input data overrides the format specification P'M' for real numbers. M is an integer.

8.2.4 Format Processing

The READ/WRITE procedures interpret a format in order to decode/encode data.

A format list contains one or more Format Items (FIs) separated by commas and enclosed by parentheses.

Example:

```
(FI1, FI2, ..., FIn)
```

FIs can be grouped into three classes. They are:

A. Form Elements (FEs) - space count S'n', line feeds R'n', or a THLL string str.

B. Action Elements (AEs) - a printed representation of a valued element indicated by I'n', O'n', H'n', D'n', P'm'F'n', E'n', L'n', or A'n', where m and n are decimal integers.

C. Repeat Elements (REs) - a sequence of one or more FIs, separated by commas, enclosed by parentheses, and prefixed by an integer number, such as:

n(FI1, FI2, ..., FIk), where n is an integer number.

An FE has a meaning by itself and is not matched up with an actual parameter in an I/O call. An AE specifies an external representation of data for an actual parameter in an I/O call. Therefore, it needs to be matched up with an actual parameter during the I/O operation. REs, finally, are merely for the convenience of the programmer and are not needed. Any formats containing REs can be expanded into an equivalent format without repeats by replacing each repeat element RE by its expansion E(RE).

The expansion E(FI) of a format list element FI is defined as follows:

$$E(FI) = \begin{cases} FI & \text{if FI is not an RE,} \\ E(FI1), \ldots, E(FIk) & \text{if } FI = n(FI1, \ldots, FIk) \text{ where } n = 1, \\ E(FI1), \ldots, E(FIk), E((n-1)(FI1, \ldots, FIk)) & \text{otherwise.} \end{cases}$$

An I/O List (IOL) is the sequence of actual parameters in a READ/WRITE procedure call after the fifth parameter. The items in this list can be expressions, array identifiers, stack identifiers, or loop arguments. This list can be thought of as being expanded, one item at a time, during I/O processing.

A loop argument is like a loop statement except that the delimiter DO and the statement following it is replaced by:

```
[a1, ..., ak]
```

where each ai is again an expression, an array identifier, a stack identifier or a loop argument. Thus, a loop argument looks like this:

```
loop control [a1, ..., ak]
```

where loop control is one of the following four constructs:

```
WHILE e
FOR var = e1  STEP e2     UNTIL e3
FOR var = e1  STEP e2     WHILE e4
FOR var = e1  REPEAT e2   WHILE e4
```

Some of the expressions in the loop control construct change, in general, after each pass through the loop. This is expressed by the following notation:

LP(o) = loop control at the beginning

LP(i) = loop control after the i'th pass through the loop.

The expansion EXP(q) of an actual parameter in IOL can now be defined as follows:

$$\text{EXP}(q) = \begin{cases} \text{q if q is an expression,} \\ \text{the ordered sequence of all subscripted variables} \\ \quad \text{within the same array if q is an array id, or} \\ \quad \text{within the same stack if q is a stack id (the} \\ \quad \text{ordering is here from the bottom to the top),} \\ \text{empty, if q is a loop argument and the terminating} \\ \quad \text{condition is met,} \\ \text{[EXP(a1), ..., EXP(ak)], EXP(LP(1)[a1, ..., ak])} \\ \quad \text{if q is loop control [a1, ..., ak]} \end{cases}$$

While executing formatted READ/WRITE procedures, the list of expanded (REPEAT free) FIs is scanned from left to right. If the next item is an FE, then it is processed without affecting the IOL. If the next item is an AE, then it is matched up with the next expression in the expanded IOL. If the format list is exhausted but the IOL is not, then scanning starts again at the beginning of the expanded format list. The process terminates when one of the following conditions is satisfied:

A. The next item in the expanded format list is an AE and the IOL is exhausted; or

B. The format list is exhausted and the IOL is exhausted.

Note that arrays are stored in memory row by row according to the scheme described in Section 3.3.

Note that stacks are stored in memory as: S(N), S(N-1), ..., S(0).

8.2.4.1 <u>WRITE Printer/Keyboard Arrays and Implied Loops</u> - The following excerpt of THLL code illustrates the use of loops to print an array. The array may also be printed by using only the array identifier.

```
        .
        .
        .
        FORMAT FID (R'1', 4(I'-2', S'1')) ;
        INTEGER ARRAY IA (1,3), BUF(70) ; INTEGER I,J ;
        POINTER PT ; DEVICE SPR = SPRINT ;
```

```
.
.
.
IA (0,0) = 00 ; IA (0,1) = 01 ; IA (0,2) = 02 ; IA (0,3) = 03 ;
IA (1,0) = 10 ; IA (1,1) = 11 ; IA (1,2) = 12 ; IA (1,3) = 13 ;
PT = OPEN (SPR, #//, 70) ;
/* THE NEXT STATEMENT PRINTS THE ARRAY USING ONLY THE
   IDENTIFIER */
WRITE (PT, LOC BUF(0), 0, FID, 0, IA) ;
/* THE NEXT STATEMENT PRINTS THE ARRAY USING A LOOP */
WRITE (PT, LOC BUF(0), 0, FID, 0, FOR I = 0 STEP 1 UNTIL 1
     [FOR J = 0 STEP 1 UNTIL 3 [IA (I,J)]]) ;
/* THE NEXT STATEMENT WILL PRINT THE VALUES ASSIGNED TO THE
     ARRAY ELEMENTS */
WRITE (PT, LOC BUF(0), 0, FID, 0, FOR I = 0 STEP 1 UNTIL 1
     [FOR J = 0 STEP 1 UNTIL 3 [I*10 + J]]) ;
```

Each WRITE statement generates two lines of output as shown below:

```
00 01 02 03
10 11 12 13
```

Note that the leading zeros were the result of using the minus sign in the field specification in the format.

8.3 INPUT/OUTPUT PROCEDURES

The THLL input/output procedures can be used to transfer information between internal storage and external devices.

8.3.1 OPEN Procedure

Each user-defined channel of communication between internal program storage and a peripheral device must be specified by the user via the pointer procedure OPEN. This procedure opens the file, creates a THLL File Control Block (THLLFCB), and returns the address of the data block to the user. The form of an OPEN procedure call is:

```
OPEN(DEV,FN,SIZE)
```

where

the returned value is the address of a THLLFCB and

DEV is the name of an I/O device.

FN is a DOUBLE variable or a string, up to eight characters long, which specifies the data file that is to be opened. Specific data file names should be used for the MDF or MTF. A null string may be used for all other devices. Whenever a string is specified, it is converted to a DOUBLE. An ALPHA variable may not be used.

SIZE is an integer expression which indicates the maximum number of words that are to be transferred by an I/O operation. For an MDF WRITE, the SIZE parameter must be the actual number of words to be transferred. For the KBDSS, the SIZE parameter must be 64. For CPRINT and SPRINT, the SIZE parameter may not exceed 1040.

The allocation of storage for the THLLFCB is machine-dependent.

BP:

Eight words for the FCB are allocated by extending the stackfrar 'or the procedure in which the call to OPEN appears. This means th this storage area is released on exit from that procedure. As a rul the user should close the file before the procedure in which the ; was opened is exited.

VAX, MC68000:

The allocation of storage to the FCB is not tied to the stackframe of the procedure calling OPEN. The life time of the FCB is from the point of creation by OPEN until its release by CLOSE or until the program ends.

The content of the FCB is machine-dependent. Details are described in References 1, 2, or 3.

Example 1:

```
DEVICE UNIT = MDF ;
POINTER P ;
P = OPEN(UNIT,#/PGMDAT/,256) ;
```

This example opens a file with the name PGMDAT on the MDF with a record length of 256 words.

Example 2:

```
DEVICE UNIT = KBDSS ;
POINTER DISP ;
DISP = OPEN(UNIT,#//,64) ;
```

This example opens a file with no name on the KBDSS with a record length of 64 words.

8.3.2 READ and WRITE Procedures

The READ and WRITE procedures cause information to be transferred in either sequential character string or binary form.

8.3.2.1 READ Procedure - The form of a READ procedure call is:

```
READ(P,PTR,OPT,FID,POSN,A1,A2, ..., An)
```

where

P	-	pointer to a THLLFCB as returned by an OPEN pointer procedure.
PTR	-	pointer to a data area which contains at least as many words as were specified by the SIZE parameter of the OPEN procedure.
OPT	-	integer expression which specifies certain read options (see Reference 1).
FID	-	format identifier or pointer expression for formatted reads of the KBDSS. This parameter should be 0 for unformatted reads of MDF, MTF, or ICL. See Section 8.2 for format declarations.
POSN	-	integer expression specifying the relative file position for reads from the MDF, MTF, or ICL. It should be 0 for formatted reads.
A1,A2, ..., An	-	actual parameters which are matched with the format list of the format declaration referenced by FID. The Ai's may be array or stack identifiers, simple, subscripted, or component variables, or loop arguments. The Ai may be omitted if the read is unformatted.

A call to READ not only reads input from the specified device, it also causes information to be put into the FCB. The information in the FCB as a result of READ is machine-dependent and is described in References 1, 2, or 3.

8.3.2.2 <u>WRITE Procedure</u> - The form of a WRITE procedure call is:

```
WRITE(P,PTR,OPT,FID,POSN,E1,E2, ..., En)
```

where

P	-	pointer to a THLLFCB as returned by an OPEN pointer procedure.
PTR	-	pointer to a data area which contains at least as many words as were specified by the SIZE parameter of the OPEN procedure.
OPT	-	integer expression specifying certain write options (see Reference 1).
FID	-	format identifier or pointer expression for formatted writes to the KBDSS, CPRINT, or SPRINT. This parameter should be 0 for unformatted writes to the MDF, MTF, or ICL. See Section 8.2 for format declarations.
POSN	-	integer expression specifying the relative file position for writes to the MDF, MTF, or ICL. It should be 0 for formatted writes.
E1,E2, ..., En	-	actual parameters which are matched with the format list of the format declaration referenced by FID. The Ei may be array or stack identifiers, expressions, or loop arguments. The Ei's may be omitted for unformatted writes.

For the KBDSS, the buffer pointed to by PTR should have at least 64 words. The buffer contains the formatted data after a WRITE to the KBDSS is completed. These data represent an image of the data being displayed. A subsequent READ from the KBDSS uses the data as presently stored in the buffer and transfer them into memory words for variables as specified in the READ statement. Therefore, to preserve the integrity of the data, the programmer should not use the buffer between a KBDSS WRITE and READ for other purposes.

A call to WRITE not only produces output on the specified device, it also causes information to be put into the FCB. The information in the FCB as a result of WRITE is machine-dependent and is described in References 1, 2, or 3.

8.3.3 CLOSE Procedure

The CLOSE procedure causes the runtime system to release the FCB from further I/O use. The form of the CLOSE procedure call is as follows:

```
CLOSE(P)
```

where

P is a pointer variable which contains the address of the THLLFCB as defined in the OPEN procedure.

8.3.4 IOWAIT Procedure

The IOWAIT procedure causes the THLL program to wait at this point until all I/O is done. When all I/O is complete, control is returned from the IOWAIT procedure and the THLL program continues its execution.

The form of the IOWAIT procedure call is as follows:

```
IOWAIT
```

CHAPTER 9

STANDARD PROCEDURES

All standard procedures in THLL are predefined; the user does not have to supply an external declaration for them. The user may, however, redefine a predefined symbol just like any user defined symbol.

Standard procedures are called like other procedures:

```
P( a1, a2, ..., ak )
```

where P is the procedure name and each ai is an actual parameter. The procedure call can only be in a statement position if P is unvalued; it can be in a statement or in an expression position if P is valued.

9.1 PREDEFINED NUMERICAL FUNCTIONS

Table 9-1 contains the list of standard numerical functions. Different types for the same argument are indicated by type abbreviations H, I, D, R, or P enclosed in braces.

The ABS function produces the absolute value of its argument.

The SIGN function determines the sign of its argument and has a positive or negative one ($\pm$1) of the appropriate type as its value. If the argument is 0, a positive 1 of the appropriate type is returned as the function value.

The SQRT function returns the square root of the argument as its value. If the argument is negative, the illegal operand fault is set on the BP.

The FLOAT functions interpret the value of the first argument as an integer and converts it into a floating point number which is the function's value. If the optional second argument is used and has a value of N, a scale factor of 2**N is applied to the function value.

The FIX functions interpret the value of the first argument as a floating point number and convert it to an integer which is the function value. For FIXI, if the optional second argument with a value of N is used with R or I type first arguments, a scale factor of 2**N is applied.

TABLE 9-1. NUMERICAL FUNCTIONS

Function Name	Type of Argument(s)	Type of Value
ABS	{H,I,D,R}	H, I, D, R correspondingly
SIGN	{H,I,D}	H, I, D (1 or -1)
	R	R (1. or -1.)
SQRT	{H,I,D,R}	R
SIN	{H,I,D,R}	R
COS	{H,I,D,R}	R
TAN	{H,I,D,R}	R
COT	{H,I,D,R}	R
ARCCOS	{H,I,D,R}	R
ARCSIN	{H,I,D,R}	R
ARCTAN	{H,I,D,R}	R
ARCCOT	{H,I,D,R}	R
LN	{H,I,D,R}	R
EXP	{H,I,D,R}	R
FLOAT	{H,I,D}	R
FLOAT	{R,I} [,I]	R
FIXH	R	H
FIXI	{R,I} [,I]	I
FIXD	R	D
SHIFTA	{H,I,D}, {H,I,D}	H, I, D corresponding to argument 1
SHIFTL	{H,I,D}, {H,I,D}	H, I, D corresponding to argument 1
SHIFTR	{H,I,D}, {H,I,D}	H, I, D corresponding to argument 1
TEST.BIT	{H,I,D}, P	I
CLR.BIT	{H,I,D}, P	I
SET.BIT	{H,I,D}, P	I
TGL.BIT	{H,I,D}, P	I
FIND.BIT	{H,I,D}, P	I
POCA	R,R,R,R	N
CAPO	R,R,R,R	N
ROAX	R,R,R,R,R	N
ROTA	R,R,R,R,R	N

Examples of FIX and FLOAT:

```
FIXI(A,I), where A real and I integer, = FIXI(A*2**I)
FIXI(J,I), where J and I integer, = SHIFTA(J,I)
FLOAT(J,I), where J and I integer, = FLOAT(J)*2**I
FLOAT(A,I), where A real and I integer, = A*2**I
```

In all cases the following identities hold:

```
FIXI(A,0) = FIXI(A), where A is of type I or R
FLOAT(A,0) = FLOAT(A), where A is of type I or R
```

The operation of FIXH, FIXI, and FIXD is machine-dependent.

BP, MC68000:

The value returned is the greatest integer less than or equal to the argument. Thus FIXI(-4.5) = -5 and FIXI(4.5) = 4.

VAX:

The value returned is the truncated argument. Thus FIXI(-4.5) = -4 and FIXI(4.5) = 4.

Of the three target machines, BP, VAX, and MC68000, the BP has the most stringent restrictions of the arguments for the trigonometric functions. Programs designed for portability should adhere to these rules for the BP. On the BP, the trigonometric functions SIN, COS, TAN, and COT are implemented with CORDIC instructions. The argument for these instructions must be in radians between -pi and pi, inclusively. In execution on the BP, if the argument is out of bounds, the illegal operand fault is set on the BP.

For the inverse trigonometric functions ARCSIN, ARCCOS, ARCTAN, and ARCCOT, the results are in radians. ARCCOT and ARCCOS return values between 0 and +pi . ARCTAN and ARCSIN return only principal values between -pi/2 and +pi/2. It is up to the programmer to implement four-quadrant capability if it is needed.

The LN function produces the natural log of the argument. If the argument is negative or zero, the illegal operand fault is set on the BP.

The EXP function produces e raised to the power specified by the argument.

For the shift functions, argument 1 is shifted according to argument 2. The absolute value of argument 2 determines how many places to shift. The shift is left if the second argument is positive and right if it is negative.

The effect of shift operations may be machine-dependent for type HALF.

BP:

A HALF operand is extended to a 32-bit INTEGER quantity and then shifted.

MC68000:

A HALF operand is shifted as a 16-bit quantity.

VAX:

HALF operands are 32-bit INTEGER quantities. Therefore, the VAX and the BP behave the same with respect to shift operations.

SHIFTA is an arithmetic shift which provides sign extension on right shifts and fills places with zeros on left shifts.

SHIFTL is a logical shift which fills places with zeros regardless of direction.

SHIFTR is a rotate shift such that the high order bits become low order bits in a left shift and vice versa for a right shift.

There are five predefined bit functions. Each takes two arguments, the first being an integer value n between 0 and 31, the second a pointer value p.

For TEST.BIT, CLR.BIT, SET.BIT, and TGL.BIT, the n'th bit in the memory word pointed to by p is affected and the condition code is set. TEST.BIT tests the bit; CLR.BIT sets the bit to 0; SET.BIT sets the bit to 1; and TGL.BIT switches the bit to 0 if it were 1 or to 1 if it were 0. In each case, the function value is the condition code as defined in Table 9-2.

TABLE 9-2. PREDEFINED BIT FUNCTION CONDITION CODES

Condition Code	Result
0	Bit n was 0
2	Bit n was 1
8	Nonexistent bit error (n LES 0 or n GRT 31)

For FIND.BIT, n designates the starting bit position from which the memory word pointed to by p is scanned while searching for the first set bit. The scan is left to right beginning at bit n. The value of FIND.BIT is defined as follows:

```
FIND.BIT(n,p) = negative integer if no bit is set,
                bit position of set bit, otherwise.
```

POCA is an unvalued function which converts from polar to Cartesian coordinates. The form of a call is:

```
POCA(r,theta,x,y)
```

where r and theta are real expressions, and x and y are real variables. Theta is an angular measure in radians. The programmer should ensure its value is between -pi and +pi. The result of POCA is to set x and y to:

```
x = r*COS(theta)
y = r*SIN(theta)
```

CAPO is an unvalued function which converts from Cartesian to polar coordinates. The form of a call is:

```
CAPO(x,y,r,theta)
```

where x and y are real expressions, and r and theta are real variables. Theta is an angular measure in radians whose returned value is between -pi and +pi. The result of CAPO is to set r and theta to the solution of the equations:

```
x = r*COS(theta)
y = r*SIN(theta)
```

Thus POCA converts polar coordinates to Cartesian coordinates and CAPO does the inverse. This means that after evaluation of:

```
POCA(e1,e2,x,y) ;
CAPO(x,y,r,theta) ;
```

the variables r and theta have the values e1 and e2, respectively. POCA is the inverse function of CAPO and vice versa.

ROAX is an unvalued function which rotates a vector specified by its Cartesian coordinates and computes the Cartesian coordinates of the rotated vector. The form of a call is:

```
ROAX(x,y,theta,x1,y1)
```

where x, y, and theta are real expressions, and x1 and y1 are real variables. Theta is an angular measure in radians. The programmer should ensure its value is between -pi and +pi. ROAX interprets x and y as:

```
x = r*COS(phi)
y = r*SIN(phi)
```

and sets x1 and y1 to:

```
x1 = r*COS(phi+theta)
y1 = r*SIN(phi+theta)
```

ROTA is an unvalued function which rotates the reference frame of a vector by an angle. The form of a call is:

```
ROTA(x,y,theta,x1,y1)
```

where x, y, and theta are real expressions, and x1 and y1 are real variables. Theta is an angular measure in radians. The programmer should ensure that the value of theta is between -pi and +pi. ROTA computes the new coordinates (x1,y1) of the vector (x,y) after rotation of the reference frame by the angle theta. This means:

ROTA(x,y,theta,x1,y1) = ROAX(x,y,-theta,x1,y1).

The POCA, CAPO, ROAX, and ROTA functions allow the solution of equations involving trigonometric expressions more efficiently than a direct approach using the standard trigonometric functions and their inverses. Examples may be found in Chapter 10.

Additional information is also available in Reference 7.

9.2 STRING FUNCTIONS

There are five functions available for string manipulation. Strings are of type A. Wherever an ALPHA variable may be used, it may be replaced by a pointer to the ALPHA variable.

The number of characters in a string is called the length of the string. A string variable (also called ALPHA variable) allows storage of strings of different length. The length of the longest string that can be stored in a string variable X is called the maximum length of X. If X is declared by ALPHA X(10), then the maximum length of X is 11. The maximum length of a string is 256 characters.

Each character in a string or ALPHA variable has a position number starting with 0 for the first character, 1 for the second, etc.

A. LENGTH(X) - This function returns the length of the string X as an integer value. A pointer to X may be used as the argument.

B. MLENGTH(X) - This function returns the maximum length of X as an integer value. A pointer to X may be used as the argument.

C. MOVEC(X,DX,Y,DY,L,V) - This function does not return a value and is assigned the type N, no type. Its effect is to move a substring of Y of length L beginning at position DY into X starting at position DX. The precise conditions that can occur and the corresponding actions are listed as follows. On the VAX and on the MC68000, the conditions are checked in the order listed. On the BP, the conditions are checked in the order 2, 3, 1, 5, 4.

Condition	Action
(1) DX $\geq$ length(X)	set illegal operand fault (on BP) and return
(2) DY $\geq$ length(Y)	set illegal operand fault (on BP) and return

(3) DY + L $\geq$ length(Y)	shorten L to length(Y) - DY
(4) DX + L $\geq$ maximum length(X)	shorten L to mlength(X) - DX
(5) DX + L $\geq$ length(X)	if V NEQ 0, shorten L to length(X) - DX, otherwise, do not shorten L
All conditions are checked	move substring and return

Thus, if V = 0, then no truncation of the moved substring to the length of the remainder string of X occurs. Omission of V is equivalent to V = 0.

The parameters DX, DY, L, and V are of type H, I, or D. The parameters X and Y may be either ALPHA variables or pointers to ALPHA variables.

D. CONC(A,X,DX,LX,Y,DY,LY) - This function is also of type N. Its effect is to form a new string A from strings X and Y by appending a substring of Y of length LY beginning at DY onto a substring of X of length LX beginning at DX. Again, the conditions that can occur and the corresponding actions are listed in the following table. On the VAX and and on the MC68000, the conditions are checked in the order listed. On the BP, the conditions are checked in the order 1, 3, 5, 2, 4, 6.

Condition	Action
(1) DX $\geq$ length(X)	set illegal operand fault (on BP) and return
(2) DY $\geq$ length(Y)	set illegal operand fault (on BP) and return
(3) DX + LX $\geq$ length(X)	shorten LX to length(X) - DX
(4) DY + LY $\geq$ length(Y)	shorten LY to length(Y) - DY
(5) LX > maximum length(A)	shorten LX to mlength(A)
(6) LX + LY $\geq$ maximum length(A)	shorten LY to mlength(A) - LX
All conditions are checked	concatenate substrings and store into A

The parameters DX, LX, DY, and LY are of type H, I, or D. The parameters A, X, and Y may be either ALPHA variables or pointers to ALPHA variables.

E. ORDERC(X,Y) - This function produces a value of type I. X and Y are strings. The value of ORDERC is:

(3) DY + L ≥ length(Y)	shorten L to length(Y) - DY
(4) DX + L ≥ maximum length(X)	shorten L to mlength(X) - DX
(5) DX + L ≥ length(X)	if V NEQ 0, shorten L to length(X) - DX, otherwise, do not shorten L
All conditions are checked	move substring and return

Thus, if V = 0, then no truncation of the moved substring to the length of the remainder string of X occurs. Omission of V is equivalent to V = 0.

The parameters DX, DY, L, and V are of type H, I, or D. The parameters X and Y may be either ALPHA variables or pointers to ALPHA variables.

D. CONC(A,X,DX,LX,Y,DY,LY) - This function is also of type N. Its effect is to form a new string A from strings X and Y by appending a substring of Y of length LY beginning at DY onto a substring of X of length LX beginning at DX. Again, the conditions that can occur and the corresponding actions are listed in the following table. On the VAX and and on the MC68000, the conditions are checked in the order listed. On the BP, the conditions are checked in the order 1, 3, 5, 2, 4, 6.

Condition	Action
(1) DX ≥ length(X)	set illegal operand fault (on BP) and return
(2) DY ≥ length(Y)	set illegal operand fault (on BP) and return
(3) DX + LX ≥ length(X)	shorten LX to length(X) - DX
(4) DY + LY ≥ length(Y)	shorten LY to length(Y) - DY
(5) LX > maximum length(A)	shorten LX to mlength(A)
(6) LX + LY ≥ maximum length(A)	shorten LY to mlength(A) - LX
All conditions are checked	concatenate substrings and store into A

The parameters DX, LX, DY, and LY are of type H, I, or D. The parameters A, X, and Y may be either ALPHA variables or pointers to ALPHA variables.

E. ORDERC(X,Y) - This function produces a value of type I. X and Y are strings. The value of ORDERC is:

-1	for X less than Y
0	for X equal to Y
1	for X greater than Y

This assumes that all characters are ordered according to the value of their numerical code on the BP and that strings are ordered lexicographically. As a special case, if X is an initial substring of Y then X precedes Y. The parameters X and Y in ORDERC(X,Y) may be ALPHA variables or pointers to ALPHA variables.

Examples:

```
ALPHA X(9),Y,Z,A,W(1),V ;
X = #*JOHN* ;              W = #*NY* ;
Y = #*JOHNATHAN* ;         V = #*NY* ;
Z = #*    JOHN* ;

LENGTH(X)              returns a value of 4
MLENGTH(Y)             returns a value of 10
ORDERC(X,Y)            returns a value of -1
ORDERC(W,Y)            returns a value of +1
ORDERC(W,V)            returns a value of 0
MOVEC(X,2,Y,0,2)       produces X = #*JOJO*
CONC(A,Z,4,4,V,0,2)    produces A = #*JOHNNY*
```

9.3 STACK FUNCTIONS

A stack is a storage area accessed by the "last in, first out" or LIFO method. Stacks must be of type H, I, D, R, or P.

PUSH(X,E) is a function which places an element on the top of stack X. E is an expression whose value is determined, converted to the type of X, and placed at the top of X. This same value is returned as the value of PUSH.

POP(X) is a function which removes or "pops" off the top element of stack X. The function value is the top element and has the type of X.

Subscripted stack identifiers form legal variables and provide another access method for stacks. However, the programmer should be sure that such a stack element exists. The occurrence of a subscripted stack identifier in the left side of an assignment statement replaces the existing stack element. It does not cause an element to be pushed onto the stack.

Before using a subscripted stack element, the programmer should establish that the element exists. The top element of a stack X is X(0) and the K'th item from the top is denoted as X(K).

STACKWC(X) is a function of type I which returns the number of entries in stack X.

STACKSC(X) is a function of type I which returns the number of unused entries in stack X.

Although each element of a double or real stack requires two memory locations, the values returned by STACKWC and STACKSC are strictly element counts, not word counts.

9.4 ARGUMENT FUNCTIONS

Four functions are provided to support optional arguments. These are arguments that do not correspond to formal parameters in the procedure being called.

```
ARGCNT
ARGPTR(N)
ARGTYPE(N)
ARGSYNCL(N)
```

ARGPTR is a pointer procedure and the other functions are integer procedures. N is an integer value identifying the N'th actual parameter of the current procedure. Details about the meanings of the function values are described in Section 6.7.

9.5 MISCELLANEOUS FUNCTIONS

BOUND(A,K) - This function is applied to an array A and returns an integer value which is the upper limit of the K'th dimension of the array A. K must be 1, 2, or 3 and is of type I. If K exceeds the declared dimension of A, then the value of BOUND is zero.

SWA(N) - N is an integer. The function value is N considered as a pointer. For the BP, that means it has base register N/2**12 and displacement N MOD 2**12. This function can be used to access known addresses in memory via components.

CHAPTER 10

EXAMPLES

This chapter contains two examples which illustrate the advantages of using the standard functions ROAX, POCA, and CAPO.

The examples are annotated to point out particular features. Both examples have been compiled so that the cross reference and symbol tables can be examined.

In the annotation, line numbers are used for references. In the examples, this appears on the left as generated in compilation.

Example: Problems Involving Polar and Cartesian Coordinates

1. If the point P has the polar coordinates (r,theta,lambda) and the Cartesian coordinates (x,y,z), then the following equations hold:

```
x = r*cos(theta)*cos(lambda)

y = r*cos(theta)*sin(lambda)

z = r*sin(theta)
```

From the polar coordinates, the Cartesian coordinates are computed by the following two statements:

```
POCA(r,theta,u,z) ;

POCA(u,lambda,x,y) ;
```

Conversely, from the Cartesian coordinates, the polar coordinates are calculated by the following two statements:

```
CAPO(x,y,u,lambda) ;

CAPO(u,z,r,theta) ;
```

2. Let (theta,lambda) be the geographic latitude and the geographic longitude of a point on Earth. Consider the problem of computing the azimuth angle A and the range angle R of a target at position (theta2,lambda2) relative to the launch point (theta1,lambda1). If D = lambda2 - lambda1, then, with the exception of two cases, the angles A and R are uniquely determined by the equations:

```
(1)  cos(R)          = sin(theta2)*sin(theta1)+cos(theta2)*cos(theta1)*cos(D)

(2)  sin(R)*cos(A)   = sin(theta2)*cos(theta1)-cos(theta2)*sin(theta1)*cos(D)

(3)  sin(R)*sin(A)   = cos(theta2)*sin(D)
```

The two exceptional cases are:

(a) theta1 = theta2, lambda1 = lambda2: launch coincides with target.

(b) theta1 = -theta2, lambda1 - lambda2 = pi: launch is antipode of target.

We use the notation:

```
x1 = sin(R)*cos(A)

y1 = sin(R)*sin(A)

z1 = cos(R)
```

According to (1) and (2), x1 and z1 are the Cartesian components of a vector that is the result of rotating the vectors with the components:

```
x = sin(theta2)

y = cos(theta2)*cos(D)
```

by the angle theta1:

```
x1 = x*cos(theta1) - y*sin(theta1)

z1 = x*sin(theta1) + y*cos(theta1)
```

x1 and z1 can be computed from x, y, theta1 by one procedure call:

```
    ROAX(x,y,theta1,x1,z1) ;
```

Thus, in order to solve the above problem, we first compute x, y, (z = y1) from theta2 and D by two calls to POCA:

```
    POCA(1,theta2,xy,x) ;

    POCA(sy,D,y,y1) ;
```

Then, x1 and z1 are computed by one call to ROAX:

```
    ROAX(x,y,theta1,x1,z1) ;
```

Finally, A and R are computed by two calls to CAPO:

```
    CAPO(x1,y1,xy1,A) ;

    CAPO(z1,xy1,ONE,R) ;
```

Giving the above calculations the form of a procedure, we obtain Example A. The listing and VAX output are shown.

Using the equations of paragraph 1, which are based on trigonometric terms, the problem can be solved. However, the solution is not as efficient. For comparison, Example B is given. The listing and VAX output are given.

Example A:

Lines 20-47:	Driving procedure TEST.
Lines 25-29:	FORMAT declarations. Three different formats are specified for use by three WRITE statements. Compare these lines with the included printed output.
Line 35:	WRITE statement which produces first line of output using FM1.
Lines 36-46:	Outer loop.
Line 38:	WRITE statement which uses FM2 and is printed each time through the outer loop.
Lines 39-44:	Inner loop.
Line 41:	Call to procedure NAVIGATE.
Line 42:	WRITE statement which uses FM3 and is printed each time through the inner loop. Note that the actual parameters to be printed are being converted from radians to degrees in the WRITE statement.
Lines 50-90:	Procedure NAVIGATE defined.
Line 51:	The four input parameters are being passed by value.
Lines 52-57:	Declarations of input parameters and comments explaining their use.
Lines 66-90:	Body of procedure NAVIGATE.
Lines 80-84:	The predefined numerical functions POCA, ROAX, and CAPO are used to solve the problem for the general case.
Line 92:	Compile unit termination.

```
EXAMPLEA                              7-JUN-1984 16:57:37  VAX-11  Tricomp/Vax  THLL5.0                              Page  1
06/11/84 15:33:10 NSWC THLL5.0      11-JUN-1984 15:09:33  UDISK1:[THLL.REFMAN.EXAMPLES]EXAMPLEA.THL;4

*    1 * EXAMPLEA                                                                                            EXAMPLEA    1
*    2 *      BEGIN                                                                                          EXAMPLEA    2
*    3 *      \\ MAIN = TEST                                                                                 EXAMPLEA    3
*    4 *                                                                                                     EXAMPLEA    4
*    5 *      SYNONYM                                                                                        EXAMPLEA    5
*    6 *           BEGIN                                                                                     EXAMPLEA    6
*    7 *           ELSEIF = / , / ;                                                                          EXAMPLEA    7
*    8 *           ENDDO = / / ;                                                                             EXAMPLEA    8
*    9 *           END ;                                                                                     EXAMPLEA    9
*   10 *                                                                                                     EXAMPLEA   10
*   11 *      OWN POINTER P ;                                                                                EXAMPLEA   11
*   12 *      OWN REAL AO,P2,PI,XO ;                                                                         EXAMPLEA   12
*   13 *      OWN INTEGER ARRAY BUFF(60) ;                                                                   EXAMPLEA   13
*   14 *                                                                                                     EXAMPLEA   14
*   15 *      PRESET AO = 1.OE-10 ;                                                                          EXAMPLEA   15
*   16 *      PRESET P2 = 1.5707963267 ;                                                                     EXAMPLEA   16
*   17 *      PRESET PI = 3.1415926535 ;                                                                     EXAMPLEA   17
*   18 *      PRESET XO = O 1E-01 ;                                                                          EXAMPLEA   18
*   19 *                                                                                                     EXAMPLEA   19
*   20 *      GLOBAL TEST ;                                                                                  EXAMPLEA   20
*   21 *      DEFINE LINK PROCEDURE TEST ;                                                                   EXAMPLEA   21
*   22 *           BEGIN                                                                                     EXAMPLEA   22
*   23 *           REAL AZ,LO1,LO2,LA1,LA2,RAD,RG ;                                                          EXAMPLEA   23
*   24 *           DEVICE OUTPUT = SPRINT ;                                                                  EXAMPLEA   24
*   25 *           FORMAT FM1(#* TARGET POINT , LAT = *,E'11',#* LONG = *,E'11',                             EXAMPLEA   25
*   26 *            R'3') ;                                                                                  EXAMPLEA   26
*   27 *           FORMAT FM2(#* LAUNCH POINT , LAT = *,E'11',S'3',#*LONG*,S'9',                             EXAMPLEA   27
*   28 *            #*AZ*,S'11',#*RG*) ;                                                                     EXAMPLEA   28
*   29 *           FORMAT FM3(S'31',E'11',S'2',E'11',S'2',E'11') ;                                           EXAMPLEA   29
*   30 *                                                                                                     EXAMPLEA   30
*   31 *           RAD = 180/PI ;                                                                            EXAMPLEA   31
*   32 *           P = OPEN (OUTPUT,#*EXAMPLEA*,32) ;                                                        EXAMPLEA   32
*   33 *           LA2 = PI/4 ;                                                                              EXAMPLEA   33
*   34 *           LO2 = O ;                                                                                 EXAMPLEA   34
*   35 *           WRITE(P,LOC BUFF(O),O,FM1,O,LA2*RAD,LO2*RAD) ;                                            EXAMPLEA   35
*   36 *           FOR LA1 = -P2 STEP PI/8 UNTIL P2+XO DO                                                    EXAMPLEA   36
*   37 *                BEGIN                                                                                EXAMPLEA   37
*   38 *                WRITE(P,LOC BUFF(O),O,FM2,O,LA1*RAD) ;                                               EXAMPLEA   38
*   39 *                FOR LO1 = -PI STEP PI/4 UNTIL PI DO                                                  EXAMPLEA   39
*   40 *                     BEGIN                                                                           EXAMPLEA   40
*   41 *                     NAVIGATE(LA1,LO1,LA2,LO2,AZ,RG) ;                                               FXAMPLEA   41
*   42 *                     WRITE(P,LOC BUFF(O),O,FM3,O,LO1*RAD,AZ*RAD,RG*RAD) ;                            EXAMPLEA   42
*   43 *                     END                                                                             EXAMPLEA   43
*   44 *                ENDDO ;                                                                              EXAMPLEA   44
*   45 *                END                                                                                  EXAMPLEA   45
*   46 *           ENDDO ;                                                                                   EXAMPLEA   46
*   47 *           END ;                   /* PROCEDURE TEST */                                              EXAMPLEA   47
*   48 *                                                                                                     EXAMPLEA   48
*   49 *                                                                                                     EXAMPLEA   49
*   50 *      DEFINE PROCEDURE NAVIGATE(LAT1,LONG1,LAT2,LONG2,A,R) ;                                         EXAMPLEA   50
*   51 *        VALUE LAT1,LONG1,LAT2,LONG2 ;                                                                EXAMPLEA   51
*   52 *        REAL LAT1 ;                /* LATITUDE OF LAUNCH POINT */                                    EXAMPLEA   52
*   53 *        REAL LONG1 ;               /* LONGITUDE OF LAUNCH POINT */                                   EXAMPLEA   53
*   54 *        REAL LAT2 ;                /* LATITUDE OF TARGET POINT */                                    EXAMPLEA   54
*   55 *        REAL LONG2 ;               /* LONGITUDE OF TARGET POINT */                                   EXAMPLEA   55
*   56 *        REAL A ;                   /* AZIMUTH ANGLE - OUTPUT */                                      FXAMPLEA   56
```

```
EXAMPLEA                          7-JUN-1984 16:57:37  VAX-11  Tricomp/Vax  THLL5.0                           Page   4
06/11/84 15:33:10 NSWC THLL5.0   11-JUN-1984 15:09:33  UDISK1:[THLL.REFMAN.EXAMPLES]EXAMPLEA.THL;4

****************************************************************************************************

 SYMBOL TABLE OF ALL SYMBOLS FOR BLOCK 6.  SON OF BLOCK 5.  SCOPE OF BLOCK IS FROM LINE 40 TO LINE 43.

****************************************************************************************************

 SYMBOL TABLE OF ALL SYMBOLS FOR BLOCK 7.  SON OF BLOCK 2.  SCOPE OF BLOCK IS FROM LINE 50 TO LINE 90.

ATTRIBUTES     OFFSET  SYMBOLS   COMMON    DEFINITIONS / REFERENCES

FRV                60  A                      50       56   /   70        83        85        86        87
VRV                40  LAT1                   50       51       52    /   71        73        82
VRV                50  LAT2                   50       51       54    /   71        73        80
VRV                48  LONG1                  50       51       53    /   69
VRV                58  LONG2                  50       51       55    /   69
FRV                64  R                      50       57   /   72        74        84

****************************************************************************************************

 SYMBOL TABLE OF ALL SYMBOLS FOR BLOCK 8.  SON OF BLOCK 7.  SCOPE OF BLOCK IS FROM LINE 66 TO LINE 90.

ATTRIBUTES     OFFSET  SYMBOLS   COMMON    DEFINITIONS / REFERENCES

SRV                68  D                      67   /   69        71        73        77       2*78       81
SRV                70  ONE                    67   /   84
SRV                78  X                      67   /   80        82
SRV                80  X1                     67   /   82        83
SRV                88  XY                     67   /   80        81
SRV                90  XY1                    67   /   83        84
SRV                98  Y                      67   /   81        82
SRV                A0  Y1                     67   /   81        83
SRV                A8  Z1                     67   /   82        84

****************************************************************************************************

 SYMBOL TABLE OF ALL SYMBOLS FOR BLOCK 9.  SON OF BLOCK 8.  SCOPE OF BLOCK IS FROM LINE 76 TO LINE 88.

****************************************************************************************************

 RUNTIME STACK REQUIREMENTS.

 BLOCK    STACK FRAME SIZE    PROCEDURE
              (DECIMAL)

    2            196            NAVIGATE
    2            180            TEST

NORMAL COMPILATION TERMINATION.
COMPILATION TIME FOR PASS1 = 388 CENTISECONDS.
COMPILATION TIME FOR PASS2 = 165 CENTISECONDS.
COMPILATION TIME FOR PASS3 = 261 CENTISECONDS.
COMPILATION TIME FOR PASS4 = 250 CENTISECONDS.
```

```
EXAMPLEA                                7-JUN-1984 16:57:37  VAX-11  Tricomp/Vax  THLL5.O                        Page   3
06/11/84 15:33:10 NSWC THLL5.O         11-JUN-1984 15:09:33  UDISK1:[THLL.REFMAN.EXAMPLES]EXAMPLEA.THL;4

 SYSTEM SYMBOL TABLE OF USED SYSTEM DEFINED SYMBOLS.  BLOCK 1.

ATTRIBUTES      OFFSET  SYMBOLS   COMMON    DEFINITIONS / REFERENCES

E P(4)                  CAPO                /  83        84
EPP(3)                  OPEN                /  32
E P(4)                  POCA                /  80        81
E P(5)                  ROAX                /  82
E P(O)                  WRITE               /  35        38        42

*****************************************************************************************************

 SYMBOL TABLE OF ALL SYMBOLS FOR BLOCK 2.  SON OF BLOCK 1.  SCOPE OF BLOCK IS FROM LINE 2 TO LINE 92.

ATTRIBUTES      OFFSET  SYMBOLS   COMMON    DEFINITIONS / REFERENCES

ORV                  8  AO                       12  /  15      2*71      2*73
OIA2(60)            30  BUFF                     13  /  35        38        42
SYNONYM(1)              ELSEIF                    7  /  73
SYNONYM(O)              ENDDO                     8  /  44        46
  P(6)                  NAVIGATE                 50  /  41
OPV                  0  P                        11  /  32        35        38        42
ORV                 10  P2                       12  /  16      2*36
ORV                 18  PI                       12  /  17        31        33        36      3*39        73        75
                                                 77     78        86
L P(O)                  TEST                     20     21
ORV                 20  XO                       12  /  18        36

*****************************************************************************************************

 SYMBOL TABLE OF ALL SYMBOLS FOR BLOCK 3.  SON OF BLOCK 2.  SCOPE OF BLOCK IS FROM LINE 21 TO LINE 47.

*****************************************************************************************************

 SYMBOL TABLE OF ALL SYMBOLS FOR BLOCK 4.  SON OF BLOCK 3.  SCOPE OF BLOCK IS FROM LINE 22 TO LINE 47.

ATTRIBUTES      OFFSET  SYMBOLS   COMMON    DEFINITIONS / REFERENCES

SRV                 40  AZ                       23  /  41        42
  F                  O  FM1                      25  /  35
  F                 3C  FM2                      27  /  38
  F                 88  FM3                      29  /  42
SRV                 48  LO1                      23  /  39        41        42
SRV                 50  LO2                      23  /  34        35        41
SRV                 58  LA1                      23  /  36        38        41
SRV                 60  LA2                      23  /  33        35        41
  D                 A8  OUTPUT                   24  /  32
SRV                 68  RAD                      23  /  31      2*35        38      3*42
SRV                 70  RG                       23  /  41        42

*****************************************************************************************************

 SYMBOL TABLE OF ALL SYMBOLS FOR BLOCK 5.  SON OF BLOCK 4.  SCOPE OF BLOCK IS FROM LINE 37 TO LINE 45.
```

```
EXAMPLEA                           7-JUN-1984 16:57:37  VAX-11  Tricomp/Vax  THLL5.0                              Page   2
06/11/84 15:33:10 NSWC THLL5.0     11-JUN-1984 15:09:33  UDISK1:[THLL.REFMAN.EXAMPLES]EXAMPLEA.THL;4

 .   57 .       REAL R ;                 /* RANGE ANGLE - OUTPUT */                                      EXAMPLEA   57
 .   58 .                                                                                                EXAMPLEA   58
 .   59 .     COMMENT LONGITUDE IS POSITIVE IF EAST , NEGATIVE IF WEST.                                  EXAMPLEA   59
 .   60 .          -PI LEQ LONG1 , LONG2 LEQ PI .                                                        EXAMPLEA   60
 .   61 .          O LEQ R LEQ PI (ALWAYS SHORTEST POSITIVE DISTANCE).                                   EXAMPLEA   61
 .   62 .          FOR EITHER POLE THE LONGITUDE CAN BE ARBITRARILY ASSUMED.                             EXAMPLEA   62
 .   63 .          IF LAUNCH IS AT POLE THEN THE OUTPUT ANGLE A MEANS THE ANGLE                          EXAMPLEA   63
 .   64 .          RELATIVE TO THE ASSUMED LONGITUDE OF THE LAUNCH POINT. ;                              EXAMPLEA   64
 .   65 .                                                                                                EXAMPLEA   65
 .   66 .          BEGIN                                                                                 EXAMPLEA   66
 .   67 .          REAL D,ONE,X,X1,XY,XY1,Y,Y1,Z1 ;                                                      EXAMPLEA   67
 .   68 .                                                                                                EXAMPLEA   68
 .   69 .          D = LONG2 - LONG1 ;                                                                   EXAMPLEA   69
 .   70 .          A = O ;                                                                               EXAMPLEA   70
 .   71 .          IF ABS(LAT2-LAT1) LEQ AO AND ABS(D) LEQ AO THEN                                       EXAMPLEA   71
 .   72 .               R = O            /* LAUNCH AT TARGET */                                          EXAMPLEA   72
 .   73 .          ELSEIF ABS(LAT2+LAT1) LEQ AO AND ABS(ABS(D)-PI) LEQ AO THEN                           EXAMPLEA   73
 .   74 .               R = PI           /* LAUNCH ANTIPODE OF TARGET */                                 EXAMPLEA   74
 .   75 .          ELSE                                                                                  EXAMPLEA   75
 .   76 .               BEGIN            /* GENERAL CASE */                                              EXAMPLEA   76
 .   77 .               IF ABS(D) GEQ PI THEN                                                            EXAMPLEA   77
 .   78 .                    D = 2*PI-ABS(D)                                                             EXAMPLEA   78
 .   79 .               IFEND ;                                                                          EXAMPLEA   79
 .   80 .               POCA(1,LAT2,XY,X) ;                                                              EXAMPLEA   80
 .   81 .               POCA(XY,D,Y,Y1) ;                                                                EXAMPLEA   81
 .   82 .               ROAX(X,Y,LAT1,X1,Z1) ;                                                           EXAMPLEA   82
 .   83 .               CAPO(X1,Y1,XY1,A) ;                                                              EXAMPLEA   83
 .   84 .               CAPO(Z1,XY1,ONE,R) ;                                                             EXAMPLEA   84
 .   85 .               IF A LES 0.0 THEN                                                                EXAMPLEA   85
 .   86 .                    A = 2.*PI + A                                                               EXAMPLEA   86
 .   87 .               ENDIF ;                                                                          EXAMPLEA   87
 .   88 .               END                                                                              EXAMPLEA   88
 .   89 .          ENDIF ;                                                                               EXAMPLEA   89
 .   90 .          END ;                 /* PROCEDURE NAVIGATE */                                        EXAMPLEA   90
 .   91 .                                                                                                EXAMPLEA   91
 .   92 .     END FINIS                                                                                  EXAMPLEA   92
 PARSE COMPLETED NORMALLY.
```

```
EXAMPLEA                             7-JUN-1984 16:57:37  VAX-11  Tricomp/Vax  THLL5.0                  Page   5
06/11/84 15:33:10 NSWC THLL5.       11-JUN-1984 15:09:33  UDISK1:[THLL.REFMAN.EXAMPLES]EXAMPLEA.THL;4

COMPILATION TIME FOR PASS5 = 172 CENTISECONDS.
LENGTH OF INTERMEDIATE CODE FILE = 198 (HEX)  408 (DECIMAL)
MAXIMUM IDSS USED =   6B (HEX)    107 (DECIMAL)
MAXIMUM SYN USED  =    1 (HEX)      1 (DECIMAL)
MAXIMUM LAB USED  =   1A (HEX)     26 (DECIMAL)
MAXIMUM ICF USED  =  198 (HEX)    408 (DECIMAL)


TRICOMP UDISK1 [THLL.REFMAN.EXAMPLES]EXAMPLEA.THL;4
```

```
TARGET POINT , LAT =  0 450E+002 LONG =  0.000E+000

LAUNCH POINT , LAT = -0.900E+002   LONG          AZ           RG
                                -0.180E+003  0.180E+003  0.135E+003
                                -0.135E+003  0.135E+003  0.135E+003
                                -0.900E+002  0.900E+002  0.135E+003
                                -0.450E+002  0.450E+002  0.135E+003
                                 0.000E+000  0.000E+000  0.135E+003
                                 0.450E+002  0.315E+003  0.135E+003
                                 0.900E+002  0.270E+003  0.135E+003
                                 0.135E+003  0.225E+003  0.135E+003
                                 0.180E+003  0.180E+003  0.135E+003
LAUNCH POINT , LAT = -0.675E+002   LONG          AZ           RG
                                -0.180E+003  0.180E+003  0.158E+003
                                -0.135E+003  0.111E+003  0.148E+003
                                -0.900E+002  0.691E+002  0.131E+003
                                -0.450E+002  0.343E+002  0.118E+003
                                 0.000E+000  0.000E+000  0.112E+003
                                 0.450E+002  0.326E+003  0.118E+003
                                 0.900E+002  0.291E+003  0.131E+003
                                 0 135E+003  0.249E+003  0.148E+003
                                 0.180E+003  0.180E+003  0.158E+003
LAUNCH POINT , LAT = -0.450E+002   LONG          AZ           RG
                                -0.180E+003  0.000E+000  0.180E+003
                                -0.135E+003  0.737E+002  0.149E+003
                                -0.900E+002  0.547E+002  0.120E+003
                                -0.450E+002  0.304E+002  0.984E+002
                                 0.000E+000  0.000E+000  0.900E+002
                                 0.450E+002  0.330E+003  0.984E+002
                                 0.900E+002  0.305E+003  0.120E+003
                                 0.135E+003  0.286E+003  0.149E+003
                                 0.180E+003  0.000E+000  0.180E+003
LAUNCH POINT , LAT = -0.225E+002   LONG          AZ           RG
                                -0.180E+003  0.951E-008  0.158E+003
                                -0.135E+003  0.473E+002  0.137E+003
                                -0.900E+002  0.473E+002  0.106E+003
                                -0.450E+002  0.306E+002  0.790E+002
                                 0.000E+000  0.000E+000  0.675E+002
                                 0.450E+002  0.329E+003  0 790E+002
                                 0.900E+002  0.313E+003  0.106E+003
                                 0.135E+003  0.313E+003  0.137E+003
                                 0.180E+003  0.951E-008  0.158E+003
LAUNCH POINT , LAT =  0.286E-008   LONG          AZ           RG
                                -0.180E+003  0.514E-008  0.135E+003
                                -0.135E+003  0.353E+002  0.120E+003
                                -0.900E+002  0.450E+002  0.900E+002
                                -0.450E+002  0.353E+002  0.600E+002
                                 0.000E+000  0.000E+000  0.450E+002
                                 0.450E+002  0.325E+003  0.600E+002
                                 0.900E+002  0.315E+003  0.900E+002
                                 0.135E+003  0.325E+003  0.120E+003
                                 0.180E+003  0.514E-008  0.135E+003
LAUNCH POINT , LAT =  0 225E+002   LONG          AZ           RG
                                -0.180E+003  0.394E-008  0.113E+003
                                -0.135E+003  0.306E+002  0.101E+003
                                -0.900E+002  0.473E+002  0.743E+002
                                -0.450E+002  0.473E+002  0.429E+002
                                 0.000E+000  0.000E+000  0 225E+002
                                 0.450E+002   0.313E+003   0.429E+002
                                 0.900E+002   0.313E+003   0.743E+002
                                 0.135E+003   0.329E+003   0.101E+003
                                 0.180E+003   0.394E-008   0.113E+003
LAUNCH POINT , LAT =  0.450E+002    LONG          AZ           RG
                                -0.180E+003   0.364E-008   0.900E+002
                                -0.135E+003   0.304E+002   0.816E+002
                                -0.900E+002   0.547E+002   0.600E+002
                                -0.450E+002   0.737E+002   0.314E+002
                                 0.000E+000   0.000E+000   0.000E+000
                                 0.450E+002   0.286E+003   0.314E+002
                                 0.900E+002   0.305E+003   0.600E+002
                                 0.135E+003   0.330E+003   0.816E+002
                                 0.180E+003   0.364E-008   0.900E+002
LAUNCH POINT , LAT =  0.675E+002    LONG          AZ           RG
                                -0.180E+003   0.394E-008   0.675E+002
                                -0.135E+003   0.343E+002   0.625E+002
                                -0.900E+002   0.691E+002   0.492E+002
                                -0.450E+002   0.111E+003   0.324E+002
                                 0.000E+000   0.180E+003   0.225E+002
                                 0.450E+002   0.249E+003   0.324E+002
                                 0.900E+002   0.291E+003   0.492E+002
                                 0.135E+003   0.326E+003   0.625E+002
                                 0.180E+003   0.394E-008   0.675E+002
LAUNCH POINT , LAT =  0.900E+002    LONG          AZ           RG
                                -0.180E+003   0.514E-008   0.450E+002
                                -0.135E+003   0.450E+002   0.450E+002
                                -0.900E+002   0.900E+002   0.450E+002
                                -0.450E+002   0.135E+003   0.450E+002
                                 0.000E+000   0.180E+003   0.450E+002
                                 0.450E+002   0.225E+003   0.450E+002
                                 0.900E+002   0.270E+003   0.450E+002
                                 0.135E+003   0.315E+003   0.450E+002
                                 0.180E+003   0.514E-008   0.450E+002
```

Example B:

Lines 20-47:	Driving procedure TEST.
Lines 25-29:	FORMAT declarations. Three different formats are specified for use by three WRITE statements. Compare these lines with the included printed output.
Line 35:	WRITE statement which produces first line of output using FM1.
Lines 36-46:	Outer loop.
Line 38:	WRITE statement which uses FM2 and is printed each time through the outer loop.
Lines 39-44:	Inner loop.
Line 41:	Call to procedure NAVIGATE.
Line 42:	WRITE statement which uses FM3 and is printed each time through the inner loop. Note that the actual parameters to be printed are being converted from radians to degrees in the WRITE statement.
Lines 50-103:	Procedure NAVIGATE defined.
Line 56:	The four input parameters are being passed by reference.
Lines 57-103:	Body of procedure NAVIGATE.
Lines 77-81:	Using assignment statements to store SINE and COSINE values in shared memory locations does not reduce execution time for THLL programs as it does for FORTRAN programs. The THLL compiler optimizes code such that this type of coding is unnecessary. However, it facilitates checkout within the procedure if such assignments are made and the variables are declared OWN.
Line 105:	Compile unit termination.

```
EXAMPLEB                            7-JUN-1984 16:57:37  VAX-11  Tricomp/Vax  THLL5.0                                      Page    1
06/11/84 15:33:37 NSWC THLL5.0     11-JUN-1984 15:32:15  UDISK1:[THLL.REFMAN.EXAMPLES]EXAMPLEB.THL;4

 *    1 * EXAMPLEB                                                                                          EXAMPLEB    1
 *    2 *      BEGIN                                                                                        EXAMPLEB    2
 *    3 *      \\ MAIN = TEST                                                                               EXAMPLEB    3
 *    4 *                                                                                                   EXAMPLEB    4
 *    5 *      SYNONYM                                                                                      EXAMPLEB    5
 *    6 *           BEGIN                                                                                   EXAMPLEB    6
 *    7 *           ELSEIF = / . / ;                                                                        EXAMPLEB    7
 *    8 *           ENDDO = / / ;                                                                           EXAMPLEB    8
 *    9 *           END ;                                                                                   EXAMPLEB    9
 *   10 *                                                                                                   EXAMPLEB   10
 *   11 *      OWN POINTER P ;                                                                              EXAMPLEB   11
 *   12 *      OWN REAL AO,P2,PI,XO ;                                                                       EXAMPLEB   12
 *   13 *      OWN INTEGER ARRAY BUFF(60) ;                                                                 EXAMPLEB   13
 *   14 *                                                                                                   EXAMPLEB   14
 *   15 *      PRESET AO = 0.1E-10 ;                                                                        EXAMPLEB   15
 *   16 *      PRESET P2 = 1.5707963267 ;                                                                   EXAMPLEB   16
 *   17 *      PRESET PI = 3.1415926535 ;                                                                   EXAMPLEB   17
 *   18 *      PRESET XO = 0.1E-01 ;                                                                        EXAMPLEB   18
 *   19 *                                                                                                   EXAMPLEB   19
 *   20 *      GLOBAL TEST ;                                                                                EXAMPLEB   20
 *   21 *      DEFINE LINK PROCEDURE TEST ;                                                                 EXAMPLEB   21
 *   22 *           BEGIN                                                                                   EXAMPLEB   22
 *   23 *           REAL AZ,LO1,LO2,LA1,LA2,RAD,RG ;                                                        EXAMPLEB   23
 *   24 *           DEVICE OUTPUT = SPRINT ;                                                                EXAMPLEB   24
 *   25 *           FORMAT FM1(#* TARGET POINT , LAT = *,E'11',#* LONG = *,E'11',                           EXAMPLEB   25
 *   26 *            R'3') ;                                                                                EXAMPLEB   26
 *   27 *           FORMAT FM2(#* LAUNCH POINT , LAT = *,E'11',S'3',#*LONG*,S'9',                           EXAMPLEB   27
 *   28 *            #*AZ*,S'11',#*RG*) ;                                                                   EXAMPLEB   28
 *   29 *           FORMAT FM3(S'31',E'11',S'2',E'11',S'2',E'11') ;                                         EXAMPLEB   29
 *   30 *                                                                                                   EXAMPLEB   30
 *   31 *           RAD = 180/PI ;                                                                          EXAMPLEB   31
 *   32 *           P = OPEN (OUTPUT,#*EXAMPLEB*,32) ;                                                      EXAMPLEB   32
 *   33 *           LA2 = PI/4 ;                                                                            EXAMPLEB   33
 *   34 *           LO2 = 0 ;                                                                               EXAMPLEB   34
 *   35 *           WRITE(P,LOC BUFF(0),0,FM1,0,LA2*RAD,LO2*RAD) ;                                          EXAMPLEB   35
 *   36 *           FOR LA1 = -P2 STEP PI/8 UNTIL P2+XO DO                                                  EXAMPLEB   36
 *   37 *                BEGIN                                                                              EXAMPLEB   37
 *   38 *                WRITE(P,LOC BUFF(0),0,FM2,0,LA1*RAD) ;                                             EXAMPLEB   38
 *   39 *                FOR LO1 = -PI STEP PI/4 UNTIL PI DO                                                EXAMPLEB   39
 *   40 *                     BEGIN                                                                         EXAMPLEB   40
 *   41 *                     NAVIGATE(LA1,LO1,LA2,LO2,AZ,RG) ;                                             EXAMPLEB   41
 *   42 *                     WRITE(P,LOC BUFF(0),0,FM3,0,LO1*RAD,AZ*RAD,RG*RAD) ;                          EXAMPLEB   42
 *   43 *                     END                                                                           EXAMPLEB   43
 *   44 *                ENDDO ;                                                                            EXAMPLEB   44
 *   45 *                END                                                                                EXAMPLEB   45
 *   46 *           ENDDO ;                                                                                 EXAMPLEB   46
 *   47 *           END ;                    /* PROCEDURE TEST */                                           EXAMPLEB   47
 *   48 *                                                                                                   EXAMPLEB   48
 *   49 *                                                                                                   EXAMPLEB   49
 *   50 *      COMMENT PROCEDURE TO CALCULATE RANGE AND AZIMUTH ANGLES FROM                                 EXAMPLEB   50
 *   51 *           LAUNCH POINT (LAT1,LONG1) TO TARGET (LAT2,LONG2).                                       EXAMPLEB   51
 *   52 *           INPUT VALUES = LAT1, LONG1, LAT2, LONG2.                                                EXAMPLEB   52
 *   53 *           OUTPUT VALUES = R, A. ;                                                                 EXAMPLEB   53
 *   54 *                                                                                                   EXAMPLEB   54
 *   55 *      DEFINE PROCEDURE NAVIGATE(LAT1,LONG1,LAT2,LONG2,A,R) ;                                       EXAMPLEB   55
 *   56 *        REAL LAT1,LONG1,LAT2,LONG2,A,R ;                                                           EXAMPLEB   56
```

```
EXAMPLEB                         7-JUN-1984 16:57:37  VAX-11  Tricomp/Vax  THLL5.0                          Page   2
06/11/84 15:33:37 NSWC THLL5.0    11-JUN-1984 15:32:15  UDISK1:[THLL.REFMAN.EXAMPLES]EXAMPLEB.THL;4

 *   57 *         BEGIN                                                                         EXAMPLEB   57
 *   58 *         REAL COSD,COSL1,COSL2,D,SINL1,SINL2 ;                                         EXAMPLEB   58
 *   59 *                                                                                       EXAMPLEB   59
 *   60 *         OWN REAL ZERO ;                                                               EXAMPLEB   60
 *   61 *         OWN REAL Y;                                                                   EXAMPLEB   61
 *   62 *                                                                                       EXAMPLEB   62
 *   63 *         PRESET ZERO = 1.0E-10 ;                                                       EXAMPLEB   63
 *   64 *                                                                                       EXAMPLEB   64
 *   65 *         D = LONG2-LONG1 ;                                                             EXAMPLEB   65
 *   66 *         A = 0.0 ;                                                                     EXAMPLEB   66
 *   67 *         IF ABS(LAT2-LAT1) LEQ ZERO AND ABS(D) LEQ ZERO THEN                           EXAMPLEB   67
 *   68 *              R = 0           /* LAUNCH AT TARGET */                                   EXAMPLEB   68
 *   69 *         ELSEIF ABS(LAT2+LAT1) LEQ ZERO AND ABS(ABS(D)-PI) LEQ ZERO                    EXAMPLEB   69
 *   70 *          THEN                                                                         EXAMPLEB   70
 *   71 *              R = 3.1415926535 /* LAUNCH ANTIPODE OF TARGET */                         EXAMPLEB   71
 *   72 *         ELSE                                                                          EXAMPLEB   72
 *   73 *               BEGIN           /* GENERAL CASE  */                                     EXAMPLEB   73
 *   74 *               IF ABS(D) GRT 3.1415926535 THEN                                         EXAMPLEB   74
 *   75 *                     D= 6.28318530 - ABS(D)                                            EXAMPLEB   75
 *   76 *               ENDIF ;                                                                 EXAMPLEB   76
 *   77 *               COSD = COS(D) ;                                                         EXAMPLEB   77
 *   78 *               SINL1 = SIN(LAT1) ;                                                     EXAMPLEB   78
 *   79 *               SINL2 = SIN(LAT2) ;                                                     EXAMPLEB   79
 *   80 *               COSL1 = COS(LAT1) ;                                                     EXAMPLEB   80
 *   81 *               COSL2 = COS(LAT2) ;                                                     EXAMPLEB   81
 *   82 *               R = SINL2*SINL1 + COSL2*COSL1*COSD ;                                    EXAMPLEB   82
 *   83 *               IF ABS(R) GRT 1.0 THEN                                                  EXAMPLEB   83
 *   84 *                    R = 1.0                                                            EXAMPLEB   84
 *   85 *               ENDIF ;                                                                 EXAMPLEB   85
 *   86 *               R = ARCCOS(R) ;                                                         EXAMPLEB   86
 *   87 *               IF R LES ZERO THEN                                                      EXAMPLEB   87
 *   88 *                    A = 0.0                                                            EXAMPLEB   88
 *   89 *               ELSE                                                                    EXAMPLEB   89
 *   90 *                  BEGIN                                                                EXAMPLEB   90
 *   91 *                  Y = (SINL2*COSL1 - COSL2*SINL1*COSD)/SIN(R) ;                        EXAMPLEB   91
 *   92 *                  IF ABS(Y) GRT 1.0 THEN                                               EXAMPLEB   92
 *   93 *                       Y = 1.0                                                         EXAMPLEB   93
 *   94 *                  ENDIF ;                                                              EXAMPLEB   94
 *   95 *                  A = ARCCOS(Y) ;                                                      EXAMPLEB   95
 *   96 *                  IF D LES 0.0 THEN                                                    EXAMPLEB   96
 *   97 *                       A = 6.28318530 - A                                              EXAMPLEB   97
 *   98 *                  ENDIF ;                                                              EXAMPLEB   98
 *   99 *                  END                                                                  EXAMPLEB   99
 *  100 *               ENDIF ;                                                                 EXAMPLEB  100
 *  101 *               END                                                                     EXAMPLEB  101
 *  102 *         ENDIF ;                                                                       EXAMPLEB  102
 *  103 *         END ;                /* PROCEDURE NAVIGATE */                                 EXAMPLEB  103
 *  104 *                                                                                       EXAMPLEB  104
 *  105 *      END FINIS                                                                        EXAMPLEB  105
 PARSE COMPLETED NORMALLY
```

```
EXAMPLEB                           7-JUN-1984 16:57:37  VAX-11  Tricomp/Vax  THLL5.O                    Page   3
06/11/84 15:33:37 NSWC THLL5.O    11-JUN-1984 15:32:15  UDISK1:[THLL.REFMAN.EXAMPLES]EXAMPLEB.THL;4

 SYSTEM SYMBOL TABLE OF USED SYSTEM DEFINED SYMBOLS.  BLOCK 1.

ATTRIBUTES     OFFSET  SYMBOLS   COMMON    DEFINITIONS / REFERENCES

EPP(3)                 OPEN                   /   32
E P(O)                 WRITE                  /   35        38        42

*******************************************************************************************************

 SYMBOL TABLE OF ALL SYMBOLS FOR BLOCK 2.  SON OF BLOCK 1.  SCOPE OF BLOCK IS FROM LINE 2 TO LINE 105.

ATTRIBUTES     OFFSET  SYMBOLS   COMMON    DEFINITIONS / REFERENCES

ORV                 8  AO                     12   /   15
DIA2(60)           30  BUFF                   13   /   35        38        42
SYNONYM(1)             ELSEIF                  7   /   69
SYNONYM(O)             ENDDO                   8   /   44        46
  P(6)                 NAVIGATE               55   /   41
OPV                 0  P                      11   /   32        35        38        42
ORV                10  P2                     12   /   16      2*36
ORV                18  PI                     12   /   17        31        33        36      3*39       69
L P(O)                 TEST                   20       21
ORV                20  XO                     12   /   18        36

*******************************************************************************************************

 SYMBOL TABLE OF ALL SYMBOLS FOR BLOCK 3.  SON OF BLOCK 2.  SCOPE OF BLOCK IS FROM LINE 21 TO LINE 47.

*******************************************************************************************************

 SYMBOL TABLE OF ALL SYMBOLS FOR BLOCK 4.  SON OF BLOCK 3.  SCOPE OF BLOCK IS FROM LINE 22 TO LINE 47.

ATTRIBUTES     OFFSET  SYMBOLS   COMMON    DEFINITIONS / REFERENCES

SRV                40  AZ                     23   /   41        42
  F                 O  FM1                    25   /   35
  F                3C  FM2                    27   /   38
  F                88  FM3                    29   /   42
SRV                48  LO1                    23   /   39        41        42
SRV                50  LO2                    23   /   34        35        41
SRV                58  LA1                    23   /   36        38        41
SRV                60  LA2                    23   /   33        35        41
  Q                A8  OUTPUT                 24   /   32
SRV                68  RAD                    23   /   31      2*35        38      3*42
SRV                70  RG                     23   /   41        42

*******************************************************************************************************

 SYMBOL TABLE OF ALL SYMBOLS FOR BLOCK 5.  SON OF BLOCK 4.  SCOPE OF BLOCK IS FROM LINE 37 TO LINE 45.

*******************************************************************************************************

 SYMBOL TABLE OF ALL SYMBOLS FOR BLOCK 6.  SON OF BLOCK 5.  SCOPE OF BLOCK IS FROM LINE 40 TO LINE 43.
```

```
EXAMPLEB                             7-JUN-1984 16:57:37  VAX-11  Tricomp/Vax  THLL5.0                    Page   4
06/11/84 15:33:37 NSWC THLL5 O      11-JUN-1984 15:32:15  UDISK1:[THLL.REFMAN.EXAMPLES]EXAMPLEB.THL;4


*****************************************************************************************************

 SYMBOL TABLE OF ALL SYMBOLS FOR BLOCK 7.  SON OF BLOCK 2.  SCOPE OF BLOCK IS FROM LINE 55 TO LINE 103.

ATTRIBUTES     OFFSET  SYMBOLS   COMMON    DEFINITIONS / REFERENCES

FRV                50  A                        55      56   /  66        88        95        97        98
FRV                40  LAT1                     55      56   /  67        69        78        80
FRV                48  LAT2                     55      56   /  67        69        79        81
FRV                44  LONG1                    55      56   /  65
FRV                4C  LONG2                    55      56   /  65
FRV                54  R                        55      56   /  68        71        82        83        84      2*86
                                                87      91

*****************************************************************************************************

 SYMBOL TABLE OF ALL SYMBOLS FOR BLOCK 8.  SON OF BLOCK 7.  SCOPE OF BLOCK IS FROM LINE 57 TO LINE 103.

ATTRIBUTES     OFFSET  SYMBOLS   COMMON    DEFINITIONS / REFERENCES

SRV                58  COSD                     58   /  77        82        91
SRV                60  COSL1                    58   /  80        82        91
SRV                68  COSL2                    58   /  81        82        91
SRV                70  D                        58   /  65        67        69        74      2*75        77        96
SRV                78  SINL1                    58   /  78        82        91
SRV                80  SINL2                    58   /  79        82        91
ORV               130  Y                        61   /  91        92        93        95
ORV               128  ZERO                     60   /  63      2*67        69        70        87

*****************************************************************************************************

 SYMBOL TABLE OF ALL SYMBOLS FOR BLOCK 9.  SON OF BLOCK 8.  SCOPE OF BLOCK IS FROM LINE 73 TO LINE 101.

*****************************************************************************************************

 SYMBOL TABLE OF ALL SYMBOLS FOR BLOCK 10  SON OF BLOCK 9.  SCOPE OF BLOCK IS FROM LINE 90 TO LINE 99.

*****************************************************************************************************

 RUNTIME STACK REQUIREMENTS

 BLOCK   STACK FRAME SIZE   PROCEDURE
             (DECIMAL)

   2            168          NAVIGATE
   2            180          TEST

NORMAL COMPILATION TERMINATION.
COMPILATION TIME FOR PASS1 = 114 CENTISECONDS.
COMPILATION TIME FOR PASS2 = 182 CENTISECONDS.
COMPILATION TIME FOR PASS3 = 252 CENTISECONDS.
```

```
EXAMPLEB                               7-JUN-1984 16:57:37  VAX-11  Tricomp/Vax  THLL5.0                              Page   5
06/11/84 15:33:37 NSWC THLL5.0        11-JUN-1984 15:32:15  UDISK1:[THLL.REFMAN.EXAMPLES]EXAMPLEB.THL;4

COMPILATION TIME FOR PASS4 = 250 CENTISECONDS.
COMPILATION TIME FOR PASS5 = 164 CENTISECONDS.
LENGTH OF INTERMEDIATE CODE FILE = 1CC (HEX)  460 (DECIMAL)
MAXIMUM IDSS USED =   6B (HEX)    107 (DECIMAL)
MAXIMUM SYN USED  =    1 (HEX)      1 (DECIMAL)
MAXIMUM LAB USED  =   21 (HEX)     33 (DECIMAL)
MAXIMUM ICF USED  =  1CC (HEX)    460 (DECIMAL)


TRICOMP UDISK1:[THLL.REFMAN.EXAMPLES]EXAMPLEB.THL;4
```

```
TARGET POINT , LAT =  0.450E+002 LONG =  0.000E+000


LAUNCH POINT , LAT = -0.900E+002   LONG          AZ            RG
                                 -0.180E+003   0.180E+003   0.135E+003
                                 -0.135E+003   0.135E+003   0.135E+003
                                 -0.900E+002   0.900E+002   0.135E+003
                                 -0.450E+002   0.450E+002   0.135E+003
                                  0.000E+000   0.000E+000   0.135E+003
                                  0.450E+002   0.315E+003   0.135E+003
                                  0.900E+002   0.270E+003   0.135E+003
                                  0.135E+003   0.225E+003   0.135E+003
                                  0 180E+003   0.180E+003   0.135E+003
LAUNCH POINT , LAT = -0.675E+002   LONG          AZ            RG
                                 -0.180E+003   0.180E+003   0.158E+003
                                  0.135E+003   0.111E+003   0.148E+003
                                 -0.900E+002   0.691E+002   0.131E+003
                                 -0.450E+002   0.343E+002   0.118E+003
                                  0.000E+000   0.000E+000   0.112E+003
                                  0.450E+002   0.326E+003   0.118E+003
                                  0.900E+002   0.291E+003   0 131E+003
                                  0 135E+003   0.249E+003   0.148E+003
                                  0.180E+003   0.180E+003   0.158E+003
LAUNCH POINT , LAT = -0.450E+002   LONG          AZ            RG
                                 -0.180E+003   0.000E+000   0.180E+003
                                 -0.135E+003   0.737E+002   0.149E+003
                                 -0.900E+002   0.547E+002   0.120E+003
                                 -0.450E+002   0.304E+002   0.984E+002
                                  0.000E+000   0.000E+000   0.900E+002
                                  0.450E+002   0.330E+003   0.984E+002
                                  0.900E+002   0.305E+003   0.120E+003
                                  0.135E+003   0.286E+003   0.149E+003
                                  0.180E+003   0.000E+000   0.180E+003
LAUNCH POINT , LAT = -0.22[illegible]02   LONG          AZ            RG
                                 -0.180E+003   0.799E-006   0.158E+003
                                 -0.135E+003   0.473E+002   0.137E+003
                                 -0.900E+002   0.473E+002   0.106E+003
                                 -0.450E+002   0.306E+002   0.790E+002
                                  0.000E+000   0.000E+000   0.675E+002
                                  0.450E+002   0.329E+003   0.790E+002
                                  0.900E+002   0.313E+003   0.106E+003
                                  0.135E+003   0.313E+003   0 137E+003
                                  0.180E+003   0.360E+003   0.158E+003
LAUNCH POINT , LAT =  0.286E-008   LONG          AZ            RG
                                 -0.180E+003   0.000E+000   0.135E+003
                                 -0.135E+003   0.353E+002   0 120E+003
                                 -0.900E+002   0.450E+002   0.900E+002
                                 -0.450E+002   0.353E+002   0.600E+002
                                  0.000E+000   0.302E-006   0.450E+002
                                  0.450E+002   0.325E+003   0.600E+002
                                  0.900E+002   0.315E+003   0.900E+002
                                  0.135E+003   0.325E+003   0.120E+003
                                  0 180E+003   0.360E+003   0.135E+003
LAUNCH POINT , LAT =  0.225E+002   LONG          AZ            RG
                                 -0.180E+003   0.000E+000   0.113E+003
                                 -0.135E+003   0 306E+002   0.101E+003
                                 -0 900E+002   0.473E+002   0.743E+002
                                 -0.450E+002   0.473E+002   0.429E+002
                                  0.000E+000   0.675E-006   0.225E+002
                                  0.450E+002   0.313E+003   0.429E+002
                                  0.900E+002   0.313E+003   0.743E+002
                                  0.135E+003   0.329E+003   0.101E+003
                                  0.180E+003   0.360E+003   0.113E+003
LAUNCH POINT , LAT =  0.450E+002     LONG          AZ           RG
                                 -0.180E+003   0.000E+000   0.900E+002
                                 -0.135E+003   0.304E+002   0.816E+002
                                 -0.900E+002   0.547E+002   0.600E+002
                                 -0.450E+002   0.737E+002   0.314E+002
                                  0.000E+000   0.000E+000   0.000E+000
                                  0.450E+002   0.286E+003   0.314E+002
                                  0.900E+002   0.305E+003   0.600E+002
                                  0.135E+003   0.330E+003   0.816E+002
                                  0.180E+003   0.360E+003   0.900E+002
LAUNCH POINT , LAT =  0.675E+002     LONG          AZ           RG
                                 -0.180E+003   0.000E+000   0.675E+002
                                 -0.135E+003   0.343E+002   0.625E+002
                                 -0.900E+002   0.691E+002   0.492E+002
                                 -0.450E+002   0.111E+003   0.324E+002
                                  0.000E+000   0.000E+000   0.225E+002
                                  0.450E+002   0.249E+003   0.324E+002
                                  0.900E+002   0.291E+003   0.492E+002
                                  0.135E+003   0.326E+003   0.625E+002
                                  0.180E+003   0.360E+003   0.675E+002
LAUNCH POINT , LAT =  0.900E+002     LONG          AZ           RG
                                 -0.180E+003   0.000E+000   0.450E+002
                                 -0.135E+003   0.450E+002   0.450E+002
                                 -0.900E+002   0.900E+002   0.450E+002
                                 -0.450E+002   0.135E+003   0.450E+002
                                  0.000E+000   0.180E+003   0.450E+002
                                  0.450E+002   0.225E+003   0.450E+002
                                  0.900E+002   0.270E+003   0.450E+002
                                  0.135E+003   0.315E+003   0.450E+002
                                  0.180E+003   0.360E+003   0.450E+002
```

CHAPTER 11

REFERENCES

1. The THLL Group, K53, <u>BP TRICOMP User's Guide</u>, NSWC TR 84-93, July 1984.

2. Jack A. Gaines, Jr., <u>MC TRICOMP User's Guide</u>, NSWC TR 84-95, July 1984.

3. John J. Zaloudek, <u>VAX TRICOMP User's Guide</u>, NSWC TR 84-97, July 1984.

4. NAVSEA OD 55658, <u>TRIDENT-II Fire Control System Mk 98 Mod 1 Support Software Programming Standards and Guidelines</u>, Nov 1983.

5. NAVSEA OD 45601, Vol 2, Part 1, <u>TRIDENT-I and TRIDENT-I Backfit Fire Control Software Real Time Operating System MONITOR Design Disclosure Document</u>, Nov 1976.

6. Hartmut G. Huber, <u>TRILOAD Reference Manual</u>, 1984. (To be published)

7. Hughes Aircraft Company, <u>TRIDENT Basic Processor Programmers Reference Manual</u>, Jul 1974.

APPENDIX A

CHARACTER SETS

A.1 THLL CHARACTER SET

The THLL character set consists of the following ASCII subset:

Uppercase letters A-Z

Numerals 0-9

The following special characters:

Character	Name	Character	Name
	Space	.	Period
!	Exclamation point	/	Slash
"	Quotation mark	:	Colon
#	Number sign	;	Semicolon
$	Dollar sign	<>	Angle brackets
%	Percent sign	=	Equal sign
&	Ampersand	?	Question mark
'	Apostrophe	@	At sign
()	Parentheses	[]	Square brackets
*	Asterisk	\	Backslash
+	Plus sign	^	Circumflex
,	Comma	_	Underline
-	Minus sign		

There is no plus or minus sign (±) defined in ASCII and the circumflex (^) is used to represent it. The TDCC devices, KBDSS, CPRINT, and SPRINT, display the circumflex as the ± character.

Not included in the THLL character set are the lowercase letters (codes X'60' - X'7E') and the nonprintable characters (codes X'00' - X'1F' and X'7E'). When these characters are used in a THLL compile unit they are treated as follows:

A. Lowercase letters are treated in a machine-dependent manner:

BP, MC68000: lowercase letters are converted to uppercase letters (code X'40' - X'5E') except when they appear in comments or remarks.

VAX: lowercase letters are converted to uppercase letters (code X'40' - X'5E') except when they appear in THLL strings or in comments or remarks.

B. Nonprintable characters are converted to question marks (?).

A.2 ASCII CHARACTER SET

The table below represents the ASCII character set. At the top of the table are hexadecimal digits (0 to 7), and to the left of the table are hexadecimal digits (0 to F). To determine the hexadecimal value of an ASCII character, use the hexadecimal digit that corresponds to the row in the "units" position, and use the hexadecimal digit that corresponds to the column in the "16's" position. For example, the value of the character representing the equal sign is 3D.

TABLE A-1. ASCII CHARACTER SET

	0	1	2	3	4	5	6	7
0	NUL	DLE	SP	0	@	P	¢	p
1	SOH	DC1	!	1	A	Q	a	q
2	STX	DC2	"	2	B	R	b	r
3	ETX	DC3	#	3	C	S	c	s
4	EOT	DC4	$	4	D	T	d	t
5	ENQ	NAK	%	5	E	U	e	u
6	ACK	SYN	&	6	F	V	f	v
7	BEL	ETB	'	7	G	W	g	w
8	BS	CAN	(	8	H	X	h	x
9	HT	EM	)	9	I	Y	i	y
A	LF	SUB	*	:	J	Z	j	z
B	VT	ESC	+	;	K	[	k	{
C	FF	FS	,	<	L	\	l	¦
D	CR	GS	-	=	M	]	m	}
E	SO	RS	.	>	N	^	n	~
F	SI	US	/	?	O	_	o	DEL

NUL	Null	DLE	Data Link Escape
SOH	Start of Heading	DC1	Device Control 1
STX	Start of Text	DC2	Device Control 2
ETX	End of Text	DC3	Device Control 3
EOT	End of Transmission	DC4	Device Control 4
ENQ	Enquiry	NAK	Negative Acknowledge
ACK	Acknowledge	SYN	Synchronous Idle
BEL	Bell	ETB	End of Transmission Block
BS	Backspace	CAN	Cancel
HT	Horizontal Tabulation	EM	End of Medium
LF	Line Feed	SUB	Substitute
VT	Vertical Tab	ESC	Escape
FF	Form Feed	FS	File Separator
CR	Carriage Return	GS	Group Separator
SO	Shift Out	RS	Record Separator
SI	Shift In	US	Unit Separator
SP	Space	DEL	Delete

APPENDIX B

TYPE MATRICES

B.1 INTRODUCTION

Every expression has a type. The meaning of the six different types provided by THLL is as follows:

HALF - an integer quantity of a machine-dependent size

BP: 16-bit quantity if the expression is a subscripted array variable, 32-bit quantity otherwise;

MC68000: 16-bit quantity; and

VAX: 32-bit quantity.

INTEGER - a 32-bit integer quantity

DOUBLE - a 64-bit integer quantity

REAL - a floating point quantity

POINTER - an address quantity

ALPHA - a character string quantity

The type of an expression is defined as follows:

Constants have the type as described in Section 2.7, variables and function values have the type as indicated in their declarations. When expressions are used as operands for some operator, the value of the resulting expression is determined from the type matrices listed below. Row headings indicate the type of the first operand, column headings indicate the type of the second operand. The operator is specified by its name. The intersection of the appropriate row and column indicates the type of the resulting expression. An asterisk in an element indicates an undefined or meaningless operation.

B.2 UNARY PLUS, UNARY MINUS

+	
H	H
I	I
D	D
R	R
P	P

-	
H	H
I	I
D	D
R	R
P	*

B.3 ADDITION, SUBTRACTION

+	H	I	D	R	P
H	H	I	D	R	P
I	I	I	D	R	P
D	D	D	D	R	P
R	R	R	R	R	*
P	P	P	P	*	*

-	H	I	D	R	P
H	H	I	D	R	*
I	I	I	D	R	*
D	D	D	D	R	*
R	R	R	R	R	*
P	P	P	P	*	I

NOTE

For the case P-P, both pointers should have values in the OWN data area or in the same group of shared variables.

B.4 MULTIPLICATION, DIVISION

*	H	I	D	R
H	H	I	D	R
I	I	I	D	R
D	D	D	D	R
R	R	R	R	R

/	H	I	D	R
H	H	I	D	R
I	I	I	D	R
D	D	D	D	R
R	R	R	R	R

B.5 MODULO AND BIT OPERATORS

NOTB	
H	H
I	I
D	D

MOD ORB ANDB XORB	H	I	D
H	H	I	D
I	I	I	D
D	D	D	D

NOTE

The MOD operator cannot be applied to real operands (type R). The operation is defined algebraically as follows:

V = A MOD B

implies A = Q*B+V

where ABS(V) < ABS(B)

and SIGN(V) = SIGN(A)

B.6 RELATIONAL OPERATORS

LES LEQ GRT GEQ	H	I	D	R	P
H	I	I	I	I	*
I	I	I	I	I	*
D	I	I	I	I	*
R	I	I	I	I	*
P	*	*	*	*	I

EQL NEQ	H	I	D	R	P	A
H	I	I	I	I	*	*
I	I	I	I	I	*	*
D	I	I	I	I	*	*
R	I	I	I	I	*	*
P	*	*	*	*	I	*
A	*	*	*	*	*	I

The integer result is represented by X'00000000' for FALSE and X'FFFFFFFF' for TRUE.

NOTES

1. Zero, representing the null pointer, can be compared with pointer values using EQL, NEQ, LES, LEQ, GRT, and GEQ.

2. ALPHA variables can be compared for equality. Two strings are considered equal if they are of equal length and the characters match one-for-one.

3. An ALPHA string can be compared to a double variable. Only the first eight characters of the string are considered. The string is considered left-justified and blank-filled. An ALPHA variable cannot be compared to a double variable.

B.7 LOGICAL OPERATORS

NOT	
H	I
I	I
D	I

AND OR XOR	H	I	D
H	I	I	I
I	I	I	I
D	I	I	I

A zero argument is interpreted as false; a nonzero argument is true. The value returned for a logical operation is X'00000000' for FALSE and X'FFFFFFFF' for TRUE.

B.8 LOC OPERATOR

LOC	
H	P
I	P
D	P
R	P
P	P
A	P

B.9 LOCA OPERATOR

LOCA	
H	I
I	I
D	I
R	I
P	I
A	I

NOTE

The executable statements, LOC and LOCA cannot operate on label, switch, common, component, device, or procedure identifiers; only simple, subscripted, or component variables or format identifiers may be used. The LOC of a procedure identifier is allowed in certain cases in presets. The LOC of a format is the address of the format, not the address of the format header. The LOC of an ALPHA is the address of the ALPHA header. The LOC of a procedure identifier is allowed only as the first argument in the PRESET functions LINKWORD and INITWORD.

B.10 EXPONENTIATION

**	H	I	D	R
H	H	I	R	R
I	H	I	R	R
D	R	R	R	R
R	R	R	R	R

The operator ** is implemented as follows:

```
X**Y =  1 if Y is 0.

X**Y =  X*(X**(Y-1)) if Y > 0 and Y is an integer.

X**Y =  (1/X)*((X)**(Y+1)) if Y < 0 and Y is an integer
        and X not equal 0.

X**Y =  EXP(Y*LN(X)) if Y is real and X > 0.

X**Y =  undefined otherwise;
                i.e., X = 0, Y < 0, Y is an integer,
                or X < 0 and Y is real.
        Effect: an illegal operand fault is generated
                on the BP.
```

NOTE

Double operands are converted to real. A warning message is issued by the compiler to consider the significance loss. A double value whose absolute value is greater than a machine-dependent maximum size yields inaccurate results.

B.11 ASSIGNMENT

The type of the assignment expression is the type of the unconverted right-side expression.

If the assignment is legal then the value *assigned to* the left-side variable is the value of the right-side expression converted to the type of the left-side variable.

The following table summarizes the conversion rules for the assignment expression.

=	H	I	D	R	P	A
H	H	H	H	H	*	*
I	I	I	I	I	*	*
D	D	D	D	D	*	*
R	R	R	R	R	*	*
P	*	*	*	*	P	*
A	*	*	*	*	*	A

NOTES

1. An exception is made for the special case of initializing or setting a pointer to the integer zero (0). The left side may be of pointer type and the right side the integer zero.

2. An exception is made for the case of a double variable on the left side and an ALPHA string on the right side. Only an ALPHA string may be used, not an ALPHA variable. The double variable contains only the first eight characters of the string, left-justified and blank-filled. The order in which the characters are stored in the double variable is machine-dependent.

APPENDIX C

TRICOMP COMPILER DIRECTIVES

C.1 GENERAL DESCRIPTION

The compiler directives are used to communicate various options to the compiler and to control the format of the printable listing.

The compiler enters the directive mode when either a \ or \\ is detected. The directive mode is terminated when the end of an input line has been detected. The \\ allows the active line to appear on the printed listing. The \ suppresses printing of the active input line. The line count indicates that a line was suppressed. The \ and \\ have no effect in a comment or SYNONYM definition. The \ or \\ can appear after any THLL item except FINIS. If during a synonym expansion a \ or \\ occurs, it causes the directive mode to be entered. The remainder of the synonym (even if on different lines in the synonym definition) and the remainder of the input line are considered to be directives. A \ detected in a synonym expansion causes that line to be suppressed. The following synonym definition allows lines containing \ to be printed:

```
SYNONYM \ = - \\ - ;
```

In a similar manner, all directives may be suppressed in the listing.

More than one directive may be included on one input line. The directives have the following general form:

```
KEY  D  Y
```

where

KEY is a keyword identifier.

D is an optional series (possibly empty) of THLL items that do not match the THLL item Y required by the KEY.

Y is either an identifier, string, number, or signed number as required by the KEY. Some directives do not require a Y.

Examples:

A. \ LINE = 1

B. \ TITLE = #'NAME OF MY TASK'

C. \\ TITLE = #'',PAGE

The first example causes the input lines to be double spaced. The second example puts a title on each page. The third example removes the user's title and causes a page to be ejected. Only the third example is printed. The = and comma are optional.

The KEY used in the directives is just a predefined identifier which has no special meaning when not in the directive mode; therefore, the KEY words are not reserved words.

When the directive mode is entered, only a KEY or end of line is recognized. After a KEY is recognized, only the Y or end of line is recognized. If the Y is a string, processing continues until the string is terminated. Therefore, the user should be careful to terminate the string. Failure to do so can cause subsequent source lines to be incorporated into the string, since string processing does not terminate at the end of the line. The directive mode is restarted after a directive has been completely recognized.

C.2 DIRECTIVE NOTATION

The following notation is used in the description of directives:

N - integer number. It may be a binary, octal, decimal or hexadecimal integer constant. The number must not be real or large enough to be considered double. An illegal number causes that directive recognition to be aborted and the directive mode to be restarted. All signs are also ignored.

SN - is an integer number with optional sign. (See the description of N above.)

S - is a string. If the string is continued on the next line, the directive is ignored and directive processing is terminated.

ID - is an identifier. The identifiers used as KEYs may also be used without causing confusion. A new directive is not started by using KEY as an identifier.

Except as noted, all directives are valid within one compile unit.

C.3 LISTING DIRECTIVES

1. ATAB SN - Adjust the TAB number by SN. If SN is negative, the tab moves to the left. The tab number does not go below 0 or above 48. Tab numbers less than 0 are replaced by 0; tab numbers greater than 48 are remembered but limited to 48.

2. HEADER - Restore the standard TRICOMP header at the top of each page. If a directive title line was being printed, it must be redefined because the page number of the last page without a header has been placed in that line. Both the standard TRICOMP header and title line would contain page numbers. HEADER is default.

3. LINE N - Print N blank lines after each printable line. Spacing does not pass the end of page. The next printable line appears at the top of the next page. Default is 0.

4. MLINE N - Print a maximum of N lines to a page. Low value of N is limited to 10. There is no upper limit value of N. Large values of N have the effect of suppressing page headers. The standard TRICOMP header, directive title line (if it exists), and the top blank line are not counted towards this line count. Default is 56.

5. NOHEADER - Suppress the standard TRICOMP header at the top of each page. If no directive title line is specified, only the home paper carriage control and page number is printed for each new page. Page number is added to the title line if it does exist, and the title line is first on the page.

6. PAGE - Immediately home paper. If this directive is printed, it would appear on the new page.

7. RSIDE N - Listing width option. Default is nonzero.

$N = 0$: The right side of the listing is suppressed. This is helpful if a listing is to be displayed on an 80 column terminal.

$N \neq 0$: Restore the right side of the listing.

The right side of the listing includes columns 73 through 90 of the input line image and the compile unit name and line number.

8. SPACE N - Print N blank lines before the next printable line. If the next printable line comes from an insert that is not being printed, then the directive is ignored. Spacing does not pass the end of page. Default is 0.

9. TAB N - Move the line image right to the Nth column. The integer number is limited to 48. Default is 0.

10. TITLE S or TITLE -S - Print the string S at the top of each page just below the standard TRICOMP header. If the minus (-) is used before the string, then the string begins in column 1 and its first character is interpreted as a carriage control character. Otherwise, the string is printed

starting in column 21. If this directive is defined in a synonym and then expanded in the S position, a maximum of 130 characters including the carriage control character can be handled by the compiler. Longer strings are truncated. If the standard TRICOMP header is suppressed, columns 121 through 130 are used to specify the page number. An empty string suppresses the directive title line. Default is no title line.

C.4 COMPILER DIRECTIVES

1. ABORT - This affects the setting of $SEVERITY at the end of TRICOMP execution (see Section G.4). If the /ALL qualifier is selected on the TRICOMP command, ABORT also has the following action. The compiler terminates the job stream if this or any of the remaining input compile units has a fatal error. All of the input THLL compile units are compiled, but after the last compile unit, TRICOMP aborts if any of the compile units has a fatal error. The directive does not cause an abort if only an earlier compile unit has a fatal error. Default is no abort.

2. BIN S - User Library name. Default is NULL. Used for configuration data. If the string contains more than eight characters, it is truncated to eight characters. This directive is meaningful only for the BP.

3. BOUNDS N - Boundcheck option. Default is nonzero.

   N = 0: No runtime check of array subscripts is performed.

   N $\neq$ 0: Runtime array bounds check. Default.

This directive alters the bounds checking at that point in the compile unit.

4. CGOPTS N - Compiler systems use. Default is 0.

5. CODEFILE ID - The default file name for the output to the assembler is the base file name of the input source file with the extension .SRC (BP), .MAR (VAX), or .MCS (MC68000). This directive changes the output file name to <ID>.DAT. Only the last file name specified by a CODEFILE directive is used for the current compile unit. The code for a single compile unit cannot be sent to more than one file.

6. CONBR N - Constant protection area base register assignment option. Default is 5. The specified base register with displacement 0 is used as the origin of the constant protection area (read only access with indirection - protection code 5). Base registers must be between 1 and 15 inclusive. This directive is meaningful only for the BP.

7. CRET - Compile as a GDDF creating compile unit. The actual GDDF is created by TRIASSM. This directive is meaningful only for the BP.

8. DEBUG N - Compiler systems use. Default is 0.

9. DSS N - D-stack size. Default is 1000. This option is used when the D-stack has overflowed the default size and the programmer has received such instruction through an error message on a previous compilation.

10. FSL N - Format size limit. Default is 512. This option should be used if error number 1*7*6 is generated by TRICOMP. In order for this directive to have an effect, it must occur before the first format declaration.

11. GDDF - Compile as a compile unit that is to be assembled using a GDDF. This directive is meaningful only for the BP.

12. HEAD N - Head-stack size. Default is 400. This option is used when the head-stack has overflowed the default size and the programmer has received such instruction through an error message on a previous compilation.

13. ICF N - ICF list size. Default is 10000. This option is used when the ICF has overflowed the default size and the programmer has received such instruction through an error message on a previous compilation.

14. ICFMIN N - The minimum number of ICF items which is written to a temporary disk file when paging the ICF. Default is 50.

15. ICFPAGE N - ICF paging option. Default is nonzero.

N = 0:	Do not write a portion of ICF to a temporary disk file, thereby keeping the entire ICF in memory.
N ≠ 0:	Write a portion of the ICF to a temporary disk file as it is being generated, the minimum block written is specified in ICFMIN. Whenever a portion of the ICF is written, memory is freed for reuse. Default.

16. IDSS N - Maximum number of identifiers. Default is 1500. This option is used when the number of identifiers has overflowed the default size and the programmer has received such instruction through an error message on a previous compilation. In order for this directive to have an effect, it must occur before the first user-defined block is opened.

17. INSBR N - Instruction protection area base register assignment option. Default is 1. The specified base register with displacement 0 is used as the origin of the instruction protection area (execute access with no indirection - protection code 6). Base registers must be between 1 and 15 inclusive. This directive is meaningful only for the BP.

18. LAB N - Compiler generated label table size. Default is 2000. This option is used when the compiler generated label table has overflowed the default size and the programmer has received such instruction through an error message on a previous compilation.

19. LIST N - List option. Default is 1.

N = 0: No listing except for errors and compile times.

N = 1: List only the source input file, errors, cross reference and compile times. Default.

N = 2: Also list the insert files in addition to the N = 1 option.

20. MAIN ID - This specifies the procedure which is the program entry point. It must be placed in the compile unit containing the procedure which is being declared the MAIN procedure. There can be only one MAIN directive per program. This directive is meaningful only for the VAX and MC68000.

21. MAXBLK N - Maximum number of blocks. Default is 400. This option is used when the number of blocks has overflowed the default size and the programmer has received such instruction through an error message on a previous compilation. In order for this directive to have an effect, it must occur before the first user-defined block.

22. NOCODE - No code is to be generated. Pass 3 and pass 4 are suppressed. This is equivalent to the PASS=2 directive.

23. NOSCHEMA - Do not generate schema data (BP only) or schema comments. Default for VAX and MC68000.

24. NOSKIP - This turns off the compiler skip mode. The SKPSTART and SKPEND directives are not honored. Default.

25. OPT N - Optimization option. Default is nonzero. Any THLL constant number can be used on the directive line.

N = 0: No optimization.

N ≠ 0: Optimization is done. Default.

This directive causes the optimization to be altered at that point in the compile unit.

26. OWNBR N - Own protection area base register assignment option. Default is 9. The specified base register with displacement 0 is used as the origin of the own protection area (read/write access with indirection - protection code 3). Base registers must be between 1 and 15 inclusive. This directive is meaningful only for the BP.

27. PASS N - Number of passes to be run. Default is 5. Pass 5 is always run.

N = 1, 2, 3, 4: Execute N passes of the compiler, and then pass 5.

N = 5: Execute all passes.

28. PRIV - Compile for privileged mode. This causes only the current compile unit to be assembled in PRIV mode. This directive is meaningful only for the BP.

29. RSS N - R-stack size. Default is 1000. This option is used when the R-stack has overflowed the default size and the programme- has received such instruction through an error message on a previous compilation.

30. SCANMAX N - The maximum number of items across which SCANBACK scans in order to optimize generated code. Default is ICF/2.

31. SCHEMA - Generate schema data (BP only) and schema comments in the generated assembly code. Default for BP.

32. SKIP - This turns on the skip mode of the compiler. All input between the SKPSTART and SKPEND directives are not processed by the compiler. The SKPSTART directive must occur after the SKIP directive. Care must be taken because skipping can go beyond compile unit boundaries.

33. SKPEND - This terminates the skipping of the input text. It only has an effect if skipping is in progress. It can come from a synonym expansion. It cannot come from an insert file that is not already active. (Insert declarations are skipped.)

34. SKPSTART - If the skip mode is active, then start skipping all input until a SKPEND directive. The SKPEND directive is the only directive that is honored.

35. SYN N - Synonym table size. Default is 1500. This option is used when the synonym table has overflowed the default size and the programmer has received such instruction through an error message on a previous compilation. In order for this directive to have an effect, it must occur before the first synonym definition.

36. TMPMAX N - Maximum number of temporaries. Default is 48. This option is used whenever a compile unit with a large stack frame size has insufficient room for temporary storage on the runtime stack and the programmer has received instruction to increase the number of temporaries through an error message on a previous compilation.

37. XREF N - This controls the level of cross reference. This directive cannot be used after the first user-defined symbol is encountered. Default is 4.

R = 0:	No symbol table or cross reference table.
R = 1:	Used system symbols and used user-defined symbols. Global symbols are included in the output. No cross reference of those symbols.
R = 2:	Used system symbols and all user-defined symbols. No cross reference of those symbols.

R = 3: Option 1 plus cross reference of those symbols.

R = 4: Option 2 plus cross reference of those symbols. Default.

R = 5: All system symbols and all user-defined symbols. Plus cross reference of those symbols.

APPENDIX D

TRICOMP COMPILER MESSAGES

D.1 INTRODUCTION

There are three classes of messages generated by the TRICOMP compiler, messages output to the user's terminal or batch log, informative messages and error messages. The messages output to the user's terminal or batch log indicate which compiler was invoked and the overall status of the compilation. The informative messages give information about the time used by each pass of the compiler and the overall success of the compilation. Error messages indicate problems found in the input text and the corrective measure (if any) attempted by the compiler. Additionally, BP TRICOMP lists the assembled object module (in binary mode) and BP related statistics.

D.2 MESSAGES OUTPUT TO THE USER'S TERMINAL OR BATCH LOG

TRICOMP uses SYS$OUTPUT to provide informative messages about the progress of a compilation. Each invocation results in three TRICOMP messages written to SYS$OUTPUT.

The first message is the following:

```
BEGIN X TRICOMP
```

where X is VAX, BP, or MC which indicates target machine.

The second message may be one of the following:

1. NORMAL COMPILATION X - indicates that the compile unit X was compiled without errors.

2. NORMAL COMPILATION X ERRORS = Y - indicates that the compile unit X was compiled with Y errors. None of the errors were fatal.

3. ABNORMAL COMPILATION X ERRORS = Y - indicates that the compile unit X was compiled with Y errors. At least one of the errors was fatal.

4. %TRICOMP-F-OPENIN, error opening X as input - indicates that the file specification X could not be opened for input.

5. %TRICOMP-F-OPENOUT, error opening X as output - indicates that the file specification X could not be opened for output.

The third message is the following:

```
END X TRICOMP
```

where X is VAX, BP, or MC which indicates target machine.

Alternatively, if there is an error in the TRICOMP command, VMS emits a %DCL-W- error message.

D.3 INFORMATIVE MESSAGES

PARSE COMPLETED NORMALLY appears just after the source listing. This message indicates that pass 2 was able to successfully recognize the executable statements in the compile unit. If any statements were incorrect, the pass 2 parser was able to recover enough in order to get to the end of the compile unit. The error messages should indicate where the statement structure is incorrect.

NORMAL COMPILATION TERMINATION or ABNORMAL COMPILATION TERMINATION appears at the end of the compilation listing. Abnormal termination indicates that an unrecoverable (fatal) error occurred and no object output is available. Normal termination means that object output is generated. The compiler output may contain errors if errors existed in the THLL input to the compiler. The error messages should indicate if there are THLL errors.

COMPILATION TIME FOR PASS X = <number> CENTISECONDS indicates the central processor time used by each pass in hundredths of seconds. Pass 1 scans the THLL compile unit, character by character, and concatenates the characters into items which are then interpreted as numbers, identifiers, and reserved words. Data declarations are also interpreted during pass 1. Pass 2 interprets THLL statements and expressions and performs the optimizations requested by the users. Passes 3 and 4 generate and format the object code output. Error messages and cross reference information are processed by pass 5.

LENGTH OF INTERMEDIATE CODE FILE = HHHH (HEX), NNNNN (DECIMAL) appears at the end of the listing of a compile unit. HHHH is the hexadecimal number indicating the number of entries needed for communication between pass 2 and pass 3 with NNNNN indicating the decimal equivalent. If the number of entries exceeds the default size then it must be increased via the ICF directive (see TRICOMP Compiler Directives, Appendix C).

MAXIMUM IDSS USED = HHHH (HEX), NNNNN (DECIMAL) is printed at the end of a THLL compile unit. HHHH is the hexadecimal number indicating the maximum number of user defined symbols or compiler predefined symbols. NNNNN is the decimal equivalent. If the number of symbols exceeds the default size then it must be increased via the IDSS directive (see TRICOMP Compiler Directives, Appendix C).

MAXIMUM SYN USED = HHHH (HEX), NNNNN (DECIMAL) is printed at the end of a THLL compile unit. HHHH is the hexadecimal number indicating the maximum number of items in a synonym right side. NNNNN is the decimal equivalent. If the maximum synonym length exceeds the default size then it must be increased via the SYN directive (see TRICOMP Compiler Directives, Appendix C).

MAXIMUM LAB USED = HHHH (HEX), NNNNN (DECIMAL) is printed at the end of a THLL compile unit. HHHH is the hexadecimal number indicating the maximum number of compiler generated labels. NNNNN is the decimal equivalent. If the number of compiler generated labels exceeds the default size then it must be increased via the LAB directive (see TRICOMP Compiler Directives, Appendix C).

MAXIMUM ICF USED = HHHH (HEX), NNNNN (DECIMAL) is printed at the end of a THLL compile unit. HHHH is the hexadecimal number indicating the maximum number of ICF entries which were kept in memory at one time. NNNNN is the decimal equivalent. If the number of maximum ICF utilization exceeds the default size then it must be increased via the ICF directive (see TRICOMP Compiler Directives, Appendix C). The effect of ICF paging can be seen by comparing this number to the length of the intermediate code file as described above.

A TWO LINE PAGE HEADER appears at the top of every TRICOMP page. The compile unit name is at the left of line 1 and mm/dd/yy and hh:mm:ss are at the left of line 2 and indicate the date and time when TRICOMP began the current compilation. This is the date and time which appears in the schema and configuration data for the current compile unit. Also, the site identification and the TRICOMP revision and change number follow the compiled date and time. The formation date and time of the compiler, its name, its revision and change number, and a three digit page number fill out the remainder of line 1. The input-file-spec and its revision date and time fill out the remainder of line 2.

EOF DETECTED is normally printed when the /ALL qualifier is specified in the TRICOMP command. It appears after the compilation of the last compile unit in the input file. If an end of file occurs before the end of a compile unit, a fatal error is reported and the compilation is immediately terminated.

D.4 ERROR MESSAGES

Error messages are printed immediately before the cross reference information. Each error message starts with the following line:

```
**** ERROR NUMBER P*MM*EE IN LINE NN OF FILE RRRRR(FFFFF).
```

where

P - is the compiler pass number 1, 2, 3, or 4.

MM - is the first half of the error number.

EE - is the second half of the error number.

NN - is the decimal line number with the file specified. NN appears as the right-most number on the source listing.

RRRRR - is the compile unit name or insert file name from an insert declaration. This name is printed just to the left of NN on the source listing.

FFFFF - is the file name specified on the TRICOMP command or the directory name from an insert declaration.

If there are more than 100 error messages, the following line is printed before the error messages:

```
<number> ERRORS DETECTED.  ONLY THE FIRST 100 WILL BE PRINTED.
```

A short explanation of the error number is printed on the lines following the error number line. If there is a symbol in error or missing and a recovery attempt is made to insert a symbol, the symbol is enclosed by a pair of single quotes. If the letter P appears in the error number, the error can occur in any pass.

P*50*1. INTERNAL COMPILER ERROR - ILLEGAL NUMBER SIZE <number>. Internal problem in the compiler. Report this error to the THLL Compiler Group.

1*2*4. ILLEGAL ITEM '<item>' FOUND IN STATEMENT. ITEM DISCARDED. Illegal item found in the input compile unit. An unimplemented special character was found within a statement. The most likely cause is a typing error or a premature end to a COMMENT. (A ; cannot appear anywhere in the COMMENT. It causes the immediate termination of the comment.) Nonprintable characters are converted to question marks (?).

1*2*5. '<number>' CAUSES A NUMERIC OVERFLOW OR UNDERFLOW. A number has been specified which cannot be represented in the target computer.

1*2*6. '<number>' CAUSES MANTISSA TRUNCATION. The number specified is too large to be represented in the target computer. TRICOMP attempts to represent the number as accurately as possible.

1*3*6. ILLEGAL ITEM '<item>' FOUND IN DECLARATION. ITEM DISCARDED. Illegal item found in a declaration. The item is probably a reserved word used only in expressions or statements.

1*3*7. ILLEGAL ITEM '<item>' BEFORE 'BEGIN'. ITEM DISCARDED. The item appears before the first BEGIN in a compile unit. Only identifiers can be accepted as a compile unit name before that BEGIN.

1*3*8. ILLEGAL ITEM '<item>' FOUND OUTSIDE DECLARATION. ITEM DISCARDED. An item that belongs within a data declaration (such as $, ICL, CPRINT, etc.) has occurred outside a data declaration. An incorrect comment could cause this error.

1*3*9. COMPILE UNIT NAME MISSING <name> ASSUMED. A compile unit name was not specified and as a result the input file name has been used as the compile unit name. The cuname indicated in the message becomes the cuname used in the schema and configuration data.

1*3*10. COMPILE UNIT '<name>' IGNORED. SPECIAL CHARACTERS ARE NOT ALLOWED IN A COMPILE UNIT NAME. A period or other special character as any one of the first eight characters in the compile unit name causes this error message. The generated code for this compile unit may not have the correct compile unit name.

1*4*7. FATAL ERROR. INCREASE SIZE OF PASS 1 PARSER STACK. A stack in the pass 1 parser has overflowed. Report this error to the THLL Compiler Group.

1*4*8. PASS 1 PARSER ERROR. FATAL ERROR. Pass 1 parser error. Report this error to the THLL Compiler Group.

1*4*9. EOF DETECTED WITHIN PROGRAM. FATAL ERROR. End of file detected within a compile unit. This is a fatal error and immediately terminates the compilation in pass 1. A missing FINIS can cause this error.

1*4*10. ILLEGAL ITEM '<item>' FOUND IN DECLARATION. ITEM DISCARDED. An item which can appear in a declaration appears in the wrong context, and the item was rejected by the error recovery algorithm and discarded.

1*4*11. NIL STRING INSERTED BEFORE '<item>' IN AN ATTEMPT TO CONTINUE. A nil string has been inserted into a declaration. This occurs when a required string is missing in a declaration and the error recovery algorithm inserted the nil string so that the pass 1 parser could continue.

1*4*12. NUMBER '0' INSERTED BEFORE '<item>' IN AN ATTEMPT TO CONTINUE. The integer 0 (zero) is inserted into a declaration. This occurs when a required number is missing in a declaration and the error recovery algorithm inserted the number 0 so that the pass 1 parser could continue. This message is also used when a format item (see Section 8.2) is needed in a format declaration.

1*4*13. '<item>' INSERTED BEFORE '<item>' IN AN ATTEMPT TO CONTINUE. An item has been inserted into a declaration. The declaration was correct up to the point indicated, but the next item was not acceptable. The pass 1 error recovery algorithm has selected an item to insert before the next item so that the parser could attempt to continue. The error recovery algorithm is a short general algorithm that considers a few special cases. Therefore, the generated symbol may not be the best choice for a particular case.

1*4*14. '<item>' DISCARDED IN AN ATTEMPT TO CONTINUE. An item has been discarded. The item was thrown out by the error recovery algorithm while backing up over a previously accepted item.

1*5*1. NEGATIVE ARGUMENT INVALID FOR <name> DIRECTIVE. The specified directive has been given a negative argument field when only a positive value is allowed. The argument is made positive.

1*5*2. INVALID <name> DIRECTIVE ARGUMENT REPLACED BY 0. An illegal numeric argument field has been specified for the named directive. The argument field is replaced by zero and the directive honored.

1*5*3. TAB COLUMN NEGATIVE. COLUMN 0 ASSUMED. The ATAB directive argument has resulted in a negative tab column. Column 0 now becomes the active tab column from this point on.

1*5*4. MLINE DIRECTIVE ARGUMENT TOO SMALL. LINE COUNT ASSUMED TO BE 10. An attempt has been made via the MLINEN directive to limit the number of lines per page below the accepted minimum. The appropriate corrective action is indicated.

1*5*5. INVALID <name> DIRECTIVE ARGUMENT IGNORED. An invalid argument has been specified for the named directive. The directive is ignored.

1*5*6. INVALID XREF DIRECTIVE ARGUMENT OR FIRST SYMBOL ALREADY ENCOUNTERED. ARGUMENT IGNORED. TRICOMP requires that the XREF directive be specified before the user symbol is supplied. Either this rule was violated or an invalid argument was presented to the XREF directive. In either case, the directive is ignored.

*5*7. INVALID FSL DIRECTIVE ARGUMENT OR FORMAT STATEMENT ALREADY ENCOUNTERED. ARGUMENT IGNORED. TRICOMP requires that the FSL directive be specified before the first format statement is encountered. Either this rule was violated or an invalid argument was presented to the FSL directive. In either case, the directive is ignored.

1*5*8. INVALID SYN DIRECTIVE ARGUMENT OR FIRST SYNONYM ALREADY ENCOUNTERED. TRICOMP requires that the directive be specified before the first synonym is encountered. Either this rule was violated or an invalid argument was presented to the SYN directive. In either case, the directive is ignored.

1*5*9. INVALID <name> DIRECTIVE ARGUMENT OR FIRST BLOCK ALREADY OPENED. ARGUMENT IGNORED. TRICOMP requires the named directive to occur before the first user-defined block. Either this rule was violated or an invalid argument was presented to the named directive. In either case, the directive is ignored.

1*5*10. ARGUMENT FIELD FOR <name> DIRECTIVE INVALID OR MISSING. The argument field for the named directive has either been discarded due to it being incorrect or has not been specified. The directive is not honored.

1*5*11. UNKNOWN <name> DIRECTIVE DISCARDED. A directive has been misspelled or an unknown character string appears in the directive field.

1*5*12. ILLEGAL ARGUMENT TYPE FOR <name> DIRECTIVE DISCARDED. An argument of the wrong syntactic class has been presented to the named directive. The argument is ignored.

1*5*13. INVALID <name> DIRECTIVE ARGUMENT. BASE REGISTER NOT IN RANGE 1 THRU 15. ARGUMENT IGNORED. The named directive has incorrectly used the specified Base Register. Only Base Registers 1 through 15 are available to THLL users. The directive is ignored.

1*7*1. MORE BEGIN'S THAN END'S. FATAL ERROR. More BEGINs than ENDs. This error is generated when a FINIS is detected and not all the blocks in the compile unit were closed. This is a fatal error which causes immediate termination of the compilation of that compile unit.

1*7*2. ILLEGAL SUBSCRIPT BOUND = <number>.
or ILLEGAL SUBSCRIPT BOUND. DOUBLE NUMBER NOT ALLOWED.
or ILLEGAL SUBSCRIPT BOUND. REAL NUMBER NOT ALLOWED.
or ILLEGAL SUBSCRIPT BOUND. ILLEGAL NUMBER TYPE = <number>.
An illegal subscript bound has been specified and that subscript bound is considered nonexistent. Either a real number or an integer number greater than 32 bits has been used as a subscript bound.

1*7*3. ILLEGAL COMPONENT OFFSET = <number>.
or ILLEGAL COMPONENT OFFSET. DOUBLE NUMBER NOT ALLOWED.
or ILLEGAL COMPONENT OFFSET. REAL NUMBER NOT ALLOWED.
or ILLEGAL COMPONENT OFFSET. ILLEGAL NUMBER TYPE = <number>.
An illegal component offset has been specified. Integers greater than a machine-dependent number and real numbers are illegal component offsets. Zero is assumed for the offset.

1*7*4. ILLEGAL COMPONENT FIELD LENGTH = <number>.
or ILLEGAL COMPONENT FIELD LENGTH. DOUBLE NUMBER NOT ALLOWED.
or ILLEGAL COMPONENT FIELD LENGTH. REAL NUMBER NOT ALLOWED.
or ILLEGAL COMPONENT FIELD LENGTH. ILLEGAL NUMBER TYPE = <number>.
An illegal component field length has been specified. Integers greater than 32 and real numbers are illegal component field lengths. A 32-bit field length is generated when this error message is generated.

1*7*5. ILLEGAL COMPONENT FIELD START BIT = <number>.
or ILLEGAL COMPONENT FIELD START BIT. DOUBLE NUMBER NOT ALLOWED.
or ILLEGAL COMPONENT FIELD START BIT. REAL NUMBER NOT ALLOWED.
or ILLEGAL COMPONENT FIELD START BIT. ILLEGAL NUMBER TYPE = <number>.
An illegal component field start bit has been specified. DOUBLES and REALS are not allowed to specify component start bit. Start bit and field length must be specified by integers. Also, the sum of start bit

and field length must be less than or equal to 32. Start bit 0 is assumed when this error message is generated.

1*7*6. FORMAT DECLARATION TOO LONG. An extremely long format declaration involving more than 511 items was processed. Only the first 511 items in the declaration were saved. Shorten the format declaration or increase the size via the FSL directive (see TRICOMP Compiler Directives, Appendix C).

1*7*8. ILLEGAL FIELD SIZE OR REPETITION COUNT IN FORMAT DECLARATION = <number>.
or ILLEGAL FIELD SIZE OR REPETITION COUNT IN FORMAT DECLARATION. DOUBLE NUMBER NOT ALLOWED.
or ILLEGAL FIELD SIZE OR REPETITION COUNT IN FORMAT DECLARATION. REAL NUMBER NOT ALLOWED.
or ILLEGAL FIELD SIZE OR REPETITION COUNT IN FORMAT DECLARATION. ILLEGAL NUMBER TYPE = <number>.
An illegal field size or repetition count appeared in a format declaration. Integers requiring more than 32 bits and real numbers cannot be used in this way.

1*7*9. IMBALANCED PARENTHESES IN FORMAT DECLARATION. Imbalanced parentheses in a format declaration. The error recovery algorithm should prevent this error message.

1*7*10. INTERNAL COMPILER ERROR - NO IDENTIFIER FOR SEMANTIC ACTION 9 PASS 1. Internal compiler error. Report this error to the THLL Compiler Group.

1*7*11. ILLEGAL INTERRUPT PROCEDURE NUMBER = <number>.
or ILLEGAL INTERRUPT PROCEDURE NUMBER. DOUBLE NUMBER NOT ALLOWED.
or ILLEGAL INTERRUPT PROCEDURE NUMBER. ILLEGAL NUMBER TYPE = <number>.
or ILLEGAL INTERRUPT PROCEDURE NUMBER. REAL NUMBER NOT ALLOWED.
The number used in the EXEC INTERRUPT PROCEDURE declaration is greater than 47 or is not an integer.

1*7*12. This message is identical to 1*10*3 in printed format. This error message occurs exclusively when there is an error concerning OPTARG. The OPTARG attribute is handled differently from other attributes. It is actually an attribute of the associated procedure rather than a formal parameter. Therefore, the error message is generated in a different manner.

1*7*13. CANNOT FIND PROCEDURE THAT USES OPTIONAL ARGUMENTS. The OPTARG attribute was being processed in pass 1 and the compiler could not find the associated procedure. This could be caused by an internal compiler error.

1*7*14. TYPE MUST APPEAR BEFORE LINK, COMPONENT, STACK OR ARRAY. One of the words INTEGER, DOUBLE, POINTER, REAL, or HALF has appeared after LINK, COMPONENT, STACK, or ARRAY. The type keyword must be first.

1*7*15. ',' INSERTED BEFORE '<id>' IN AN ATTEMPT TO CONTINUE. A comma was missing in a list of identifiers or before OPTARG in a procedure argument list.

1*7*16. VALUE PART MUST OCCUR FIRST IN A PROCEDURE HEAD. When describing the formal parameters in a procedure declaration, those parameters passed by value must be specified in a VALUE declaration before specifying the type for any of the formal parameters.

1*7*17. MISSING ',' INSERTED IN THIS DECLARATION IN AN ATTEMPT TO CONTINUE. A comma was missing between the elements of a format or component declaration.

1*7*18. THE PROPER FORM FOR THIS DECLARATION IS 'EXEC INTERRUPT <NUM.> PROCEDURE <ID.>'. Either EXEC or INTERRUPT was missing in the procedure head of this procedure declaration.

1*7*19. 'LOGICAL' or 'ARITHMETIC' MUST APPEAR BEFORE 'FIELD'. Either LOGICAL or ARITHMETIC appeared after FIELD in the component declaration.

1*7*20. '<id>' NOT ALLOWED IN FORMAT DECLARATION. A typing error resulted in the use of an identifier within a format list element.

1*7*21. ILLEGAL NUMBER IN FORMAT = <number>.
or ILLEGAL NUMBER IN FORMAT. DOUBLE NUMBER NOT ALLOWED.
or ILLEGAL NUMBER IN FORMAT. REAL NUMBER NOT ALLOWED.
or ILLEGAL NUMBER IN FORMAT. ILLEGAL NUMBER TYPE = <number>.
A number type other than the special number types for formats was used within a format. The only acceptable number types for formats are R, S, A, I, O, F, E, H, D, L, and P. Regular number types can only be used as a repetition count. The regular number types are decimal and real numbers and the number types C, X, and B.

1*7*22. '$' IGNORED IN AN ATTEMPT TO CONTINUE. A $ has appeared before an R number (carriage return count) in a format declaration. This construct is illegal.

1*7*23. P.NUM IGNORED IN AN ATTEMPT TO CONTINUE. P.NUMs are used to place the decimal point when using an F output format in a format declaration. This error occurs when P.NUM is used with other output formats.

1*7*24. MISSING 'DEFINE' IN PROCEDURE DECLARATION. All procedure declarations must begin with DEFINE. It is not allowed in a procedure head when describing a formal parameter as a procedure. DEFINE is also prohibited in EXTERNAL declarations.

1*7*25. COMMONS MUST NOT OVERLAP. PREVIOUS COMMON CLOSED. The previous common was not terminated by an ENDCOM or COMEND. An ENDCOM is assumed and the previous common is terminated.

1*7*26. SUPERFLUOUS ENDCOM OR COMEND. COMMON NOT OPEN. An ENDCOM or COMEND reserved word has occurred. Either no common has been started or an

ENDCOM or COMEND already followed the last common declared in this block. This can occur when attempting to nest commons within the same block. Commons defined in different blocks do not overlap, even when the blocks are nested. The last common in a block is assumed closed at the end of the block if the ENDCOM or COMEND is missing.

1*7*27. '<attribute>' MAY NOT BE DEFINED IN A COMMON. Only simple variables, ALPHA variables, arrays, and stacks may be included in a common. Devices, formats, components, labels, switches, and procedures may not be included in commons. GLOBAL, EXTERNAL, and OWN may not be specified in the common body.

1*9*1. MAXIMUM NUMBER OF ALLOWED BLOCKS EXCEEDED. FATAL ERROR. Maximum number of allowed blocks exceeded. This is a pass 1 fatal error that causes immediate termination of the compilation for the current compile unit. Increase the number of blocks allowed via the MAXBLK compiler directive (see TRICOMP Compiler Directives, Appendix C).

1*9*2. MORE END'S THAN BEGIN'S. FATAL ERROR. This error is a fatal error that causes the immediate termination of the compilation for the current compile unit. This error should not occur because the error recovery algorithm should have discarded the extra ENDs.

1*9*3. '<id>' INCOMPLETELY DEFINED. The identifier in this error message is not completely defined. The most likely cause for this error message is that an identifier is listed in the argument list of a procedure declaration and does not appear in the specification part of the procedure head.

1*10*1. INCREASE SIZE OF PASS 1 ID STACK. FATAL ERROR. An internal stack in pass 1 has overflowed. Increase the internal stack size via the IDSS compiler directive (see TRICOMP Compiler Directives, Appendix C).

1*10*2. ILLEGAL DEFINITION <attribute> FOR IDENTIFIER '<id>'. DEFINITION WILL BE DISCARDED. An illegal combination of attributes has been initially specified for an identifier. The identifier exists in the compiler but none of the specified attributes are saved. If the specified attributes do not agree with the associated data declaration, then consult the THLL Compiler Group.

1*10*3. NEW DEFINITION <attribute> IS INCONSISTENT WITH OLD DEFINITION <attribute> FOR IDENTIFIER '<id>'. NEW DEFINITION WILL BE DISCARDED. The same identifier in the same block has appeared in contradictory data declarations. The cross reference should indicate the lines where the identifier was previously defined. A misunderstanding of the compile unit's block structure could cause this error.

1*10*4. '<id>' MULTIPLY DEFINED AS <attribute>. The same identifier in the same block has appeared in contradictory data declarations. The multiply defined attributes are listed in this error message. A misunderstanding of the block structure of the compile unit could cause this error. The attributes from the current line are ignored.

1*10*5. SHARED IDENTIFIER '<id>' FOUND OUTSIDE PROCEDURE, OWN ADDED. A non-OWN identifier was defined in the outermost block (block 2). Variables, arrays, and stacks must be declared as OWN.

1*10*6. '<id>' WAS NOT PREVIOUSLY DEFINED AS A FORMAL PARAMETER. DEFINITION IGNORED. An error concerning the formal parameters in a procedure declaration has occurred. The description of the formal parameters which appears before the procedure body includes the description of an identifier not specified in the formal parameter list. The extra identifier is not added to the formal parameter list. The formal parameter list is the list of identifiers enclosed in parentheses which appears just after the procedure identifier.

1*10*7. GLOBAL '<id>' NOT DECLARED OWN. OWN ADDED. A non-OWN simple variable, ALPHA, stack, or array was declared GLOBAL. The OWN attribute has been added in an attempt to retain meaningful information.

1*10*8. FORMAL PARAMETER '<id>' CANNOT BE PASSED BY VALUE. VALUE IGNORED. An array, stack, ALPHA, format, or device identifier was included in the value part of the procedure head. None may be passed by value. The corresponding identifier should be removed from the value statement.

1*10*9. '<id>' CANNOT BE DECLARED GLOBAL. GLOBAL IGNORED. The <id> has been declared global in an earlier declaration. Devices, switches, labels, and components cannot be declared global. The global attribute for that <id> was dropped.

1*10*10. '<id>' PREVIOUSLY DEFINED OUTSIDE COMMON. NOT INCLUDED IN COMMON. The <id> has occurred in a declaration in the same block before the beginning of the common. If the <id> appears in a declaration after the ENDCOM or COMEND, another error message identifies the multiple declaration.

1*10*11. GLOBAL PROCEDURE '<id>' MUST BE DECLARED IN BLOCK 2. Any global procedure must be declared in the outermost block, not local to a procedure.

1*10*12. PROCEDURE '<id>' CANNOT BE A FORMAL PARAMETER FOR A LINK PROCEDURE. No procedure entry point may be passed to a link procedure. When a link procedure is called, the base registers can be reloaded. Therefore, the new address space may not be valid for the parameter procedure.

1*11*1. SYNONYM BODY TABLE OVERFLOW. FATAL ERROR. The synonym table has overflowed. Increase the synonym table size via the SYN compiler directive (see TRICOMP Compiler Directives, Appendix C).

1*11*2. ILLEGAL SYNONYM NAME 'XX...X'. 'XX...X' is the spelling of the item.

1*11*3. EQUAL (=) MISSING IN SYNONYM DECLARATION. The equal (=) has been omitted in a synonym declaration. The equal is added.

1*11*4. 'XX...X' ENCOUNTERED IN A SYNONYM DECLARATION. A ';' IS EXPECTED. 'XX...X' is the spelling of the item.

1*11*5. SYNONYM 'XX...X' CAN'T BE CALLED CYCLICALLY. 'XX...X' is the spelling of the item.

1*11*6. MULTIPLE LEVELS OF INSERT CALLS NOT ALLOWED. FATAL ERROR. Insert calls can appear only in the text of the original source text. Insert calls cannot appear within an insert file.

2*1*2. FATAL ERROR. UNEXPECTED END OF FILE. Internal compiler error. Report this error to the THLL Compiler Group.

2*1*3. UNEXPECTED CONSTANT BEING REMOVED IN AN ATTEMPT TO CONTINUE. Two adjacent operands are found. The second is removed in an attempt to continue. An out of place data declaration can also cause this message.

2*1*5. FATAL ERROR. INCREASE HEAD (HEAD STACK). The Head stack has overflowed. Increase it over the current size via the HEAD compiler directive (see TRICOMP Compiler Directives, Appendix C).

2*1*6. FATAL ERROR. INCREASE DSS (D STACK). The D stack has overflowed. Increase it over the current size via the DSS compiler directive (see TRICOMP Compiler Directives, Appendix C).

2*1*7. FATAL ERROR. INCREASE RSS (R STACK). The R stack has overflowed. Increase it over the current size via the RSS compiler directive (see TRICOMP Compiler Directives, Appendix C).

2*1*8. FATAL ERROR. INCREASE LAB (LABEL TABLE). The compiler generated label table has overflowed. Increase it over the current size via the LAB compiler directive (see TRICOMP Compiler Directives, Appendix C).

2*1*9. FATAL ERROR. INCREASE ICF (ICF LIST). The ICF working buffer has overflowed. Increase it over the current size via the ICF compiler directive (see TRICOMP Compiler Directives, Appendix C).

2*1*10. COMMON '<id>' CANNOT OCCUR IN A STATEMENT OR EXPRESSION. Symbols within a COMMON block can be referenced, the origin of the COMMON cannot.

2*1*11. FATAL ERROR. TOO MANY UNDEFINED IDENTIFIERS. TRICOMP allows only 50 undefined identifiers.

2*1*12. THE ARGUMENT HAS AN ILLEGAL TYPE FOR THE CURRENT OPERATOR. An argument of the wrong type has been presented to an operator.

2*1*13. LABEL '<id>' IS NOT A VALID TARGET FOR BLOCK EXIT.
or NO VALID TARGET FOR BLOCK EXIT.
The label <id> used in an EXIT statement is not a label for a currently active block; or the user cannot exit from the current block.

2*1*14. LABEL '<id>' IS NOT A VALID TARGET FOR LOOP EXIT.
or NO VALID TARGET FOR LOOP EXIT.
The label <id> used in a LOOPEXIT statement is not a label for a currently active loop; or the user cannot exit from the current loop; or the user may not be in a loop.

2*1*15. SWITCH INDEX MUST BE AN INTEGER EXPRESSION.
or IF TEST MUST BE AN INTEGER EXPRESSION.
or CASE TEST MUST BE AN INTEGER EXPRESSION.
or WHILE TEST MUST BE AN INTEGER EXPRESSION.
or ARRAY SUBSCRIPT MUST BE AN INTEGER EXPRESSION.
or STACK SUBSCRIPT MUST BE AN INTEGER EXPRESSION.
or OR OPERAND MUST BE AN INTEGER EXPRESSION.
or AND OPERAND MUST BE AN INTEGER EXPRESSION.
or XOR OPERAND MUST BE AN INTEGER EXPRESSION.
or NOT OPERAND MUST BE AN INTEGER EXPRESSION.
The index for a case statement, switch, stack subscript, array subscript, or logical operator is an illegal type for the named operation.

2*1*16. COMPONENT '<id>' MUST USE A POINTER EXPRESSION. The expression for a component must be a pointer expression.

2*1*17. PROCEDURE '<id>' HAS TOO MANY ACTUAL PARAMETERS. The call for a procedure contains too many parameters.

2*1*18. ACTUAL PARAMETER NUMBER '<number>' IN CALL FOR '<id>' IS INVALID. The n'th actual parameter's type is not compatible with the n'th formal parameter's type.

2*1*19. ARRAY '<id>' HAS WRONG NUMBER OF SUBSCRIPTS. The use of an array contains the wrong number of subscripts.

2*1*20. SWITCH INDEX HAS BEEN CONVERTED TO INTEGER FROM REAL.
or IF TEST HAS BEEN CONVERTED TO INTEGER FROM REAL.
or CASE TEST HAS BEEN CONVERTED TO INTEGER FROM REAL.
or WHILE TEST HAS BEEN CONVERTED TO INTEGER FROM REAL.
or ARRAY SUBSCRIPT HAS BEEN CONVERTED TO INTEGER FROM REAL.
or STACK SUBSCRIPT HAS BEEN CONVERTED TO INTEGER FROM REAL.
or OR OPERAND HAS BEEN CONVERTED TO INTEGER FROM REAL.
or AND OPERAND HAS BEEN CONVERTED TO INTEGER FROM REAL.
or XOR OPERAND HAS BEEN CONVERTED TO INTEGER FROM REAL.
or NOT OPERAND HAS BEEN CONVERTED TO INTEGER FROM REAL.
The index for a case statement, switch, stack subscript, array subscript, or logical operator has been converted from a real expression to an integer expression.

2*1*21. ARRAY '<id>' HAS CONSTANT INDEX OUT OF BOUNDS. A constant index has been specified which is greater than the range of the named array.

2*1*22. PASS 2 COMPILER ERROR (1). An internal compiler error has occurred. Report this error to the THLL Compiler Group.

2*1*23. '<id>' CAN BE USED ONLY IN PRESETS. LINKWORD and INITWORD are only PRESET functions.

2*1*24. INVALID USE FOR INLINE FUNCTION '<id>'. The named function is not syntactically correct in its use.

2*1*25. THE OPERATOR '<id>' IS INCORRECTLY USED IN A PRESET. The operator along with its operand(s) is not a supported preset expression.

2*1*26. NON-OWN VARIABLE '<id>' CAN NOT BE PRESET. Only OWN variables may be preset in THLL.

2*1*27. INCONSISTENT RANGE VARIABLES '<number>' AND '<number>' IN LEFT SIDE OF PRESET. Presetting may take place in the storage for only one variable at a time.

2*1*28. INVALID PRESET RANGE. The scope for a preset went outside the storage area for a variable, or it was not possible to end the preset at the user-specified location.

2*1*29. ALPHA VARIABLE '<id>' MAY NOT BE RANGE PRESET. It is not possible to range preset ALPHA variables.

2*1*30. OVERLAPPING PRESET FOR VARIABLE '<id>' NOT VALID. It is not possible to preset a storage element more than once.

2*1*31. TOO MANY RIGHT SIDE PRESET EXPRESSIONS. For a range preset, there are more preset expressions than storage elements. For a simple preset, there are more preset expressions than storage elements.

2*1*32. COMPLETE VALUE NOT FOUND FOR VARIABLE OR EXPRESSION. The value for a variable or expression is not known at this time. No presetting is done.

2*1*33. INVALID USE OF IDENTIFIER '<id>' IN PRESETS. Labels, stacks, and switches may not be used in presets in any way. LOC of own and external variables is allowed. Formal parameters and shared variables are not allowed.

2*1*34. ATTEMPT TO PRESET BEYOND DECLARED RANGE OF ALPHA VARIABLE '<id>'. An ALPHA variable has been preset with more characters than the declaration allows. The truncated string is assigned to the variable.

2*1*35. RETURN EXPRESSION IGNORED FOR PROCEDURE THAT HAS NO TYPE. Untyped procedures should return with the statement 'RETURN' rather than with 'RETURN expression'.

2*1*36. SWITCH INDEX HAS BEEN CONVERTED TO INTEGER FROM DOUBLE.
or CASE TEST HAS BEEN CONVERTED TO INTEGER FROM DOUBLE.
or ARRAY SUBSCRIPT HAS BEEN CONVERTED TO INTEGER FROM DOUBLE.
or STACK SUBSCRIPT HAS BEEN CONVERTED TO INTEGER FROM DOUBLE.
This is a warning message. Only the least significant half of the double expression is used in the named operation. This message does not mean

that an error exists, but it is used to flag possible problems if the most significant half of double expression is significant.

2*1*37. '**' OPERAND HAS BEEN CONVERTED TO REAL FROM DOUBLE. This is a warning message. It is used to flag possible significance loss when a double exceeds the significance of a real.

2*1*38. UNDEFINED '<id>' ASSUMED TO BE A SWITCH. Identifier is undefined. It is assumed to be a switch identifier in an attempt to continue.

2*1*39. UNDEFINED '<id>' ASSUMED TO BE A LABEL. Identifier is undefined. It is assumed to be a label identifier in an attempt to continue.

2*1*40. UNDEFINED '<id>' ASSUMED TO BE AN INTEGER PROCEDURE. Identifier is undefined. It is assumed to be an integer procedure in an attempt to continue.

2*1*41. UNDEFINED '<id>' ASSUMED TO BE AN INTEGER VARIABLE. Identifier is undefined. It is assumed to be an integer variable in an attempt to continue.

2*1*42. COMPONENT '<id>' CANNOT BE INDEXED. The component does not have a field specification which can be indexed (see Section 3.6.5).

2*1*43. COMPONENT '<id>' MUST BE EITHER REGULAR OR INDEXED. The component has more than two arguments.

2*1*44. THE OPERATION '<operator> <type>' IS INVALID AND HAS BEEN CHANGED TO '<operator> <type>' IN AN ATTEMPT TO CONTINUE. The operand type for the unary operator is invalid.

2*1*45. THE OPERATION '<type> <operator> <type>' IS INVALID AND HAS BEEN CHANGED TO '<type> <operator> <type>' IN AN ATTEMPT TO CONTINUE. The operand types for the binary operator are invalid.

2*1*46. FATAL ERROR. STACK UNDERFLOW. This error message may be caused by errors in the source compile unit such as too many END's.

2*1*47. PROCEDURE '<id>' MUST BE DEFINED IN PRIVILEGED MODE. EXEC and EXEC INTERRUPT procedures must be defined in privileged mode. See the PRIV directive in TRICOMP Compiler Directives, Appendix C.

2*1*48. A RETURN IN A VALUED PROCEDURE MUST HAVE A RETURN EXPRESSION. This is a warning message. A zero of the appropriate type is supplied as the default return value.

2*1*49. LINK, EXEC, OR EXEC INTERRUPT PROCEDURE '<id>' CANNOT BE A PARAMETER IN A PROCEDURE CALL. EXEC and EXEC INTERRUPT procedures have special entry sequences. Link procedures must be called with a LINK instruction. The procedure being called cannot distinguish these parameter procedures from normal parameter procedures.

2*1*50. LOC OF PROCEDURE '<id>' IS USED IN THE WRONG CONTEXT. The LOC of a procedure identifier can be used only as the first parameter of LINKWORD or INITWORD on the right side of a preset.

2*1*51. THE <IF EXPRESSION> OR <CASE EXPRESSION> USED IN THIS CONTEXT MUST NOT HAVE A COMMON TYPE <NULL>. The lowest inclusive common type is type N and the IF or CASE construct is being used for its value. Pointer or ALPHA values are generally incompatible with other types. Check the type tables for details.

2*1*52. PROCEDURE '<id>' HAS TOO FEW ACTUAL PARAMETERS.

2*2*1. SYNTAX ERROR, LAST ITEM READ '<item>'.

2*2*4. SYNTAX ERROR, LAST ITEM READ '<item>'.
SEMICOLON INSERTED.

2*2*5. SYNTAX ERROR, LAST ITEM READ '<item>'.
DISCARDED '<item>'.
or SYNTAX ERROR, LAST ITEM READ '<item>'.
DISCARDED '<item>' AND ITEM PRECEDING IT.
The item is always an operator or a delimiter. The second version indicates that a symbol or a constant preceding the operator/delimiter has been discarded also.

2*2*6. SYNTAX ERROR, LAST ITEM READ '<item>'.
MISSING '<item>'.

2*2*7. SYNTAX ERROR, LAST ITEM READ '<item>'.
SUPERFLUOUS '<item>'.

2*2*8. SYNTAX ERROR, LAST ITEM READ '<item>'.
'<item>' REPLACED BY FINIS.

2*2*9. SYNTAX ERROR, LAST ITEM READ '<item>'.
NO EXPRESSIONS OR STATEMENTS ALLOWED IN OUTERMOST BLOCK.
If the outermost block contains expressions or statements, then this error message appears on the END closing the outermost block.

2*2*10. SYNTAX ERROR, LAST ITEM READ '<item>'.
MOST RECENT <element> REPLACED BY 0.
<element> may be one of the following:

- PROGRAM
- BLOCK
- PARENTHESIZED EXPRESSION
- IF EXPRESSION
- CASE EXPRESSION

2*2*11. SYNTAX ERROR, LAST ITEM READ '<item>'.
MOST RECENT <kind> DECLARATION DISCARDED.
<kind> may be one of the following:
STATEMENT OR
PRESET
PROCEDURE

2*2*12. FINIS ENCOUNTERED IN WRONG CONTEXT.

2*2*13. TOO MANY ERRORS, PARSER GIVES UP. This message is issued when 100 syntax errors have been encountered.

2*2*14. LOST IN ERRORS, HEAD STACK UNDERFLOW. This message may be issued if the outermost block contains expressions or statements and there is a semicolon before FINIS.

2*2*15. SYNTAX ERROR, LAST ITEM READ '<item>'.
LOST IN ERRORS, IL STACK OVERFLOW.
An internal table overflows. Most likely this will never happen. If so, correct the errors and try it again.

2*2*16. SYNTAX ERROR, LAST ITEM READ '<item>'.
PRESET MODE WILL BE TERMINATED.
The indicated syntax error causes the termination of preset mode.

2*2*18. COMPILER ERROR ERRP2. An internal compiler error has occurred in Procedure ERRP2. Report this error to the THLL Compiler Group.

3*1*1 through 3*1*20. An internal compiler error has occurred during code generation. Report these errors to the THLL Compiler Group.

3*1*21. INCREASE TMPMAX PARAMETER. NO TEMPS AVAILABLE AT CODE FILE POSITION = <number>. Due to a large stack frame size, the compiler was unable to allocate sufficient runtime stack size for temporaries. Increase the temporary stack allocation size via the TMPMAX compiler directive (see TRICOMP Compiler Directives, Appendix C).

3*1*22. VAL$ OPERATOR IMPROPERLY USED AT CODE FILE POSITION = <number>. The VAL$ operator is used in VAX TRICOMP to change the normal mode for passing arguments to procedures written in other languages. THLL programs always use the "VAX pass by reference" calling mechanism, but procedures written in other languages may need to receive parameters by the "VAX pass by value" method. VAL$ is only allowed in VAX THLL programs, and it can only be used on actual parameters during a procedure call. VALUE must be specified for the corresponding parameter description in the EXTERNAL declaration or procedure definition of the procedure being called.

4*1*1. COMPATIBILITY DATA TABLE OVERFLOW. The table which contains compatibility information for this compile unit has overflowed its default size. Report this error to the THLL Compiler Group.

4*1*2. BASE REGISTER <number> USED BY SYSTEM AND OWN DATA.
FATAL ERROR.
or BASE REGISTER <number> USED BY SYSTEM AND INSTRUCTIONS.
FATAL ERROR.
or BASE REGISTER <number> USED BY SYSTEM AND CONSTANTS.
FATAL ERROR.
or BASE REGISTER <number> USED BY OWN DATA AND INSTRUCTIONS.
FATAL ERROR.
or BASE REGISTER <number> USED BY OWN DATA AND CONSTANTS.
FATAL ERROR.
or BASE REGISTER <number> USED BY INSTRUCTIONS AND CONSTANTS.
FATAL ERROR.
The indicated base register is currently spanning the two indicated protection areas which is illegal. This situation can occur when one protection area becomes large enough to require the next consecutive base register, which happens to be allocated, or when base registers are indeed multiply allocated via the OWNBR, INSBR, or CONBR directives (see TRICOMP Compiler Directives, Appendix C).

D.5 COMPILER ASSEMBLED OBJECT LISTING (BP TRICOMP ONLY)

When the Binary Mode of compilation is invoked, either by default or the /BIN qualifier on the TRICOMP command, the compiler does not produce an assembly source code file. Instead, it assembles the assembly source code for the TRIDENT Basic Processor (TBP) and produces a binary code file.

A listing of the assembled object is produced between the THLL source and the cross reference map.

The format of this listing is a line-by-line list of the assembly object where each line is prefixed by a header. The header for a line of assembly object which assembles to a value (i.e., not ASMOPT, GLOBAL, ENTRY, SECT, or END lines) is:

```
XXXX YYYYYYYY PQ
```

where YYYYYYYY is an assembled value at virtual address XXXX. P and Q are either the letters A, E, L, or a blank. The letter A implies a relocatable address, E implies an external address, and L implies an LTO. If a letter appears in the P position, then the left halfword is meant. If it appears in the Q position, then the right halfword is meant. The header is all blank for other assembly lines.

D.6 COMPILER GENERATED STATISTICS (BP TRICOMP ONLY)

A one page summary of statistics is printed following the listing of the THLL source code and before the symbol table and cross reference map.

This summary of the instruction mix appears in five groups of three columns in each group. Each group contains the opcode, the absolute count of this opcode and the percentage of this opcode in the total instruction mix. The width for each group of data is 24 characters. The format for the data in a group is shown below:

```
------    ------     --.---
OP CODE MNEMONIC  DECIMAL COUNT  PERCENTAGE OF USE
```

These 24 lines are followed by two blank lines. Next is data concerning the configuration summary. The summary includes the site identification, NSWC or GEOS, the revision and change number of TRICOMP as well as the TRILOAD library name, the name of the current compile unit, and the date and time when the compilation started. The format of the configuration summary is as follows:

```
CONFIGURATION SUMMARY
TRICOMP <site> THLL<rev>.<change>
CU LIBR,<librname> NAME,<cuname> DATE,<mm/dd/yy> TIME,<hh.mm.ss>
```

The configuration summary is followed by a blank line. Next is data concerning the various protection areas used. The virtual origin of a protection area is provided in hexadecimal. The size of that protection area is given in both hexadecimal and decimal. If an area is not used, there is no printout for that area. The three possible lines of output are:

```
PC3 ORIGIN <virtual address> OWN SIZE <number> HEX <number> DEC
PC6 ORIGIN <virtual address> INSTRUCTION SIZE <number> HEX <number> DEC
PC5 ORIGIN <virtual address> CONSTANT SIZE <number> HEX <number> DEC
```

The protection area summary is followed by a blank line and then an optional summary of the compiler predefined routines used by this compile unit. All these routines reside on the TRILOAD System Library, SYSLIB. Secondary routines (required by these referenced predefined routines) do not appear in this list, but they are automatically loaded by TRILOAD. In some cases, more than one predefined procedure is included in one SYSLIB library module. If the user redefines one of these predefined procedures by an external declaration or procedure declaration, then the name does not appear in this list because the system predefined procedure is not used. The format of this optional summary is as follows:

```
SYSTEM DEFINED LIBRARY ROUTINES USED BY THIS PROGRAM
<name1>  <name2>  ...
```

These data lines are followed by a blank line. The next line indicates the compiler mode. This line is either:

```
        SOURCE MODE
or
        BINARY MODE
```

APPENDIX E

SYMBOL TABLE AND CROSS REFERENCE MAP EXPLANATIONS

E.1 EXPLANATION OF SYMBOL TABLE

The symbol table is organized in alphabetical order within each block. The blocks are listed in the order in which they are opened. The system predefined symbols are placed into block 1, which by definition encloses all user-defined blocks. User-defined blocks start with block number 2. The parent block (enclosing block) number of each block is printed in the header of the symbol table for that block. The scope of the block is also indicated by embracing line numbers. These line numbers refer to those on the left of the listing.

A block has two forms. Any BEGIN END pair delimits a block. Since a block can contain other blocks, an equal number of BEGINs and ENDs must be contained between the BEGIN END pair. The other form of block is called a formal parameter block. A formal parameter block starts just after the procedure identifier in a procedure declaration, and it ends at the end of the procedure body.

Example:

```
DEFINE INTEGER PROCEDURE X
                          /* Formal parameter block begins */
       (A,B);
       INTEGER A, B;

          BEGIN
             .
             . (procedure body)
             .
          END            /* Formal parameter block ends */
```

NOTE

A formal parameter block is opened even when there are no formal parameters.

Symbol table information is printed below the following header.

```
ATTRIBUTES     OFFSET  SYMBOLS   COMMON
```

The ATTRIBUTES column is broken up into a number of fields. The first field indicates the storage class. The second field indicates the type of the identifier. The third field represents the kind of data structure. If the symbol is a synonym, then SYNONYM is printed starting in the first of these columns. An optional fourth field indicates the size (in THLL words) of an array header. An optional fifth field (which is the fourth field if the array header size is not present) indicates additional information based on the kind of symbol.

The first field indicates the storage class. The attributes below are listed in the priority used in selecting the printed attribute. Therefore, something that is both global and OWN has the global attribute printed.

V - formal parameter passed by value
F - formal parameter passed by reference
L - LINK - either EXTERNAL or GLOBAL
G - global
E - external
O - OWN
S - shared - non-OWN arrays, stacks and simple variables
b - blank, none of the above

The second field indicates the type associated with the symbol.

H - type half
I - type integer
D - type double
R - type real
P - type pointer
A - type ALPHA
B - used as block label
F - used as FOR label
E - EXEC procedure
T - EXEC INTERRUPT procedure
b - blank - untyped (type N)

The third field indicates the kind or syntactic class of the symbol. The following attributes are possible.

V	- simple variable
A	- array
S	- stack
P	- procedure
D	- device
F	- format
L	- label
W	- switch
C	- component
M	- common

The fourth field indicates the size of an array header.

2	- one-dimensional array header
4	- two-dimensional array header
6	- three-dimensional array header

NOTE

This field is not used if the header does not exist. In the case of no headers, the fifth field becomes the fourth field.

The fifth field varies depending on the kind of symbol.

(d1,d2,d3)	- Present for commons, arrays, stacks, and alphas. (d1,d2,d3) indicates the size of each dimension of the data item. (d1,d2,d3) will be (d1) for a common, one dimensional array, stack or alpha; (d1,d2) for a two dimensional array; and (d1,d2,d3) for a three dimensional array. d1, d2, and d3 may be *.
((sb,nb),off)	- Present for components. ((sb,nb),off) indicates the start bit, number of bits and the offset of the component. (sb,nb) is not present for an ALPHA component.
(par)	- Present for procedures. (par) indicates the number of formal parameters that the procedure has. If a procedure has optional arguments, then (par,OPTARG) indicates this.
(swi)	- Present for switches. (swi) indicates the number of switch elements that the switch has.

The OFFSET column is printed as a hexadecimal number. It is used for a number of purposes. For all OWN and COMMON variables, arrays, and stacks, the runtime storage allocation relative to the origin of the proper program section appears in this field. For shared variables, arrays, and stacks, the number represents the offset within the compiler supplied runtime stack frame. The offset for an array is for the first element of the array. The offset for FORMATS, ALPHA variables, and stacks is the address of the header. For the BP, offsets are in units of words. For the VAX and MC68000, they are in units of bytes.

The SYMBOL column contains the first 19 characters of the identifier. If the identifier belongs to a common, then the SYMBOL column contains the first 9 characters of of the identifier, a space, and the first 9 characters of the common name.

E.2 EXPLANATION OF CROSS REFERENCE MAP

When a cross reference map is requested, the symbol information appears on the left part of the page. The line numbers where the identifier is defined appear first to the right of the symbol information, followed by a "/", and then the line numbers where the identifier is used. All line numbers are decimal and refer to the numbers on the right side of the TRICOMP listing.

The references appear in the following form:

```
[N*]LLLL[,I]
```

where

N	- the number of times the identifier is referenced on that line. The [N*] is not printed when N = 1.
LLLL	- the line number within the input file.
I	- the insert file that the line comes from. If the line comes from the input file specified on the TRICOMP command, the [,I] is not printed.

If a symbol is undefined or defined in a block of insufficient scope, the symbol is included in the symbols of block 0, which appears after the last user-defined block. If there are no undefined symbols, the block 0 output is suppressed.

If a cross referenced symbol occurs in an insert file, the name of that insert file can be found in the list of insert file names at the end of the cross reference listing. The format for the insert file summary is as follows:

```
LIST OF INSERT FILES

NN  RRRRRRRR(FFFFFFFF)
 .
 .
 .
```

where

NN - the insert file number used in the references above

RRRRRRRR - the insert file name

FFFFFFFF - the insert file directory name as explained in Appendix G.

After the undefined symbols (block 0), and the list of insert files (if they exist), a summary of the commons appears. The base offset of the common and the GLOBAL or EXTERNAL attribute of the common can be found in the symbol table for the block where the common is declared.

The summary for each common starts with the following line:

```
SUMMARY FOR COMMON AAAAAAAA SIZE OF COMMON (QQQQQ) = XX (HEX) DD (DEC)
```

where

AAAAAAAA is the name of the common

(QQQQQ) is (WORDS) for BP TRICOMP, and (BYTES) for VAX and MC TRICOMP

XX is the size of the common as a hexadecimal number

DD is the size of the common as a decimal number

Information concerning symbols in a common is printed in a table of six columns. The table may be broken into three parts spread across the page.

Adjacent symbols in the common appear vertically in the symbol section. The symbol at the end of the first part is adjacent with the symbol at the top of the second part. The table has the following two-line header (repeated three times across the page).

QQQQ OFST (HEX)	SYMBOL	HDR SIZE	SIZE (DEC)	(QQQQQ) (HEX)

where

QQQQ is WORD for BP TRICOMP, and BYTE for VAX and MC TRICOMP

(QQQQQ) is (WORDS) for BP TRICOMP, and (BYTES) for VAX and MC TRICOMP

The symbol name within the common appears in the SYMBOLS column. The hexadecimal offset of that symbol from the beginning of the common is given in the OFST column. This is the offset to the first word after the header of arrays, stacks, and ALPHAs. If the symbol has a header, the size of the header is printed as a decimal number. The size of the data structure is calculated (excluding headers) and printed as hexadecimal and decimal numbers. Allocation of memory is target machine-dependent. Therefore, differences in the length of commons and in the size of some data structures should be expected for different target machines.

The following modified example was generated for the BP. It would appear broken into three similar columns.

WORD OFST (HEX)	SYMBOL	HDR SIZE	SIZE (WORDS) (DEC)	(HEX)
0	F1		2	2
3	ALPH	1	1	1
4	G1		1	1
5	EI		1	1
6	DV1		2	2
8	DV2		2	2
A	DV3		2	2
10	RAG	4	30	1E
32	MOP	4	42	2A
5E	PSS	2	21	15
73	COSE		1	1
74	PIE		1	1
75	FRIEND		1	1

A summary of the runtime stack requirements for the compile unit appears after the symbols for block 0, the insert file list, and the COMMON summary (if they exist). A stack frame is acquired from the runtime stack whenever a procedure is entered. The stack frame is sufficient to handle all storage required by shared variables, arrays, and stacks defined within the procedure, formal parameters, compiler-generated temporaries, and system support routine scratch areas. Certain system procedures such as OPEN extend the current stack frame. This extension is not included in the runtime stack requirements. Only GLOBAL or referenced procedures are listed in the summary since only those procedures can contribute to the runtime stack. The summary is not printed if the \ XREF = 0 directive (suppress symbol table and cross reference output) is used.

If no callable procedures exist in the compile unit, the following line is printed:

```
NO RUNTIME STACK REQUIREMENTS.
```

The runtime stack summary has the following form:

```
RUNTIME STACK REQUIREMENTS.

BLOCK   STACK FRAME SIZE   PROCEDURE
            (DECIMAL)

  2            26            PROC1
  2            20            PROC2
```

The block number indicates the block in which the procedure is defined. The stack frame size is expressed as a decimal integer. For BP TRICOMP, the stack frame size is in words, whereas for MC and VAX TRICOMP, it is in bytes. The structure of the runtime stack and the layout of the stack frames is target machine-dependent.

APPENDIX F

THLL PROGRAM REPORTS

F.1 GLOBAL CROSS REFERENCE REPORT

The global cross reference report presents an overview of the global symbols of a program. The actual individual line references for those symbols (as well as symbols local to a compile unit) can be found in the local cross reference at the end of the .TLS file. The global cross reference report is useful in determining which procedures and modules reference a particular global symbol. Symbols defined within an insert file or within a COMMON are not global, but are normally used in many compile units and are therefore included in the global cross reference report.

F.1.1 Producing a Global Cross Reference Report

The notation .EXT; *implies the highest version for input* and the creation of a new version for output.

The global cross reference report is developed from the global cross reference data files that result when TRICOMP compiles a compile unit. These files by default have a .GXR; extension. This report is generated in a two-step process.

The first step is to combine the .GXR; files and the current master cross reference data file into a new master cross reference data file. The master cross reference data file by default has a .MXD; extension. Updating the master cross reference data file is done by invoking the GXRUPDATE command. By default GXRUPDATE executes under the assumption that all .GXR; files to be combined reside in the current default directory. The following command invokes GXRUPDATE:

```
$ GXRUPDATE[/qualifier] file-spec
```

where file-spec is the master cross reference data file. If file-spec is just Filename, then by default GXRUPDATE combines the .GXR; files and the current Filename.MXD; file into a new Filename.MXD; file.

GXRUPDATE has the following command qualifier:

1. /GXR=(file-spec[,...]) - This requests that only the selected files be used to update the master cross reference data file.

The second step is to transform the master cross reference data file into a readable global cross reference report. This is done by invoking GXRREPORT with the following command.

```
$ GXRREPORT[/qualifier...] file-spec
```

where filespec is the master cross reference data file produced by the previous step. The file produced by GXRREPORT is the master cross reference report, which by default has a .MXR; extension. This report can be examined and/or printed.

GXRREPORT has the following two command qualifiers:

1. /TITLE="The User's LEQ 36 Character Title" - This requests that "The User's LEQ 36 Character Title" be placed on the top of each page of the report. The default title is "Global Cross Reference Report".

2. /MXR=mxr-file-spec - This requests that the global cross reference report be placed in mxr-file-spec. Default is Filename.MXR.

This two-step process allows incremental updates to the global cross reference data. As the need arises to change selected compile units, the .THL files are changed, compiled using TRICOMP, assembled, and linked. Later, the global cross reference data files are gathered by invoking GXRUPDATE to do a partial update to the master global cross reference data file. GXRREPORT is used to form the new report. The changed compile units is reflected accordingly in the new report.

The master global cross reference data file is a running record of the global references in a program. It is important not to delete this file. Each invocation of GXRUPDATE and GXRREPORT creates a new version of the master global cross reference data and master global cross reference report files respectively.

F.1.2 Reading a Global Cross Reference Report

All GLOBAL symbols and symbols declared within insert files are contained in the Global Cross Reference Report. The symbols are arranged alphabetically in the report. The first four lines on each page of the Global Cross Reference Report contain the following header.

```
GLOBAL CROSS REFERENCE The User's LEQ 36 Character Title      PAGE  1

SYMBOL    ATTRIBUTES    DEFINED        USED IN    USED IN    REFERENCES
 NAME                   IN DECK          DECK    PROCEDURE  USED  CHNG
```

The SYMBOL NAME column contains up to 9 characters of the identifier. VAX TRICOMP and MC TRICOMP support 31 character significance so sometimes the entire identifier cannot be placed in this column. This is known as the nine character restriction. This restriction is as follows: If the identifier contains less than 10 characters, then the entire identifier appears in the SYMBOL NAME column. If the identifier contains more than 9 characters, then the first 8 characters followed by an asterisk appear in the SYMBOL NAME column.

The ATTRIBUTES column contains a short description of the identifier (listed in the SYMBOL NAME column). The ATTRIBUTES column in the Global Cross Reference Report corresponds to the ATTRIBUTES column in the symbol table. See the definition of the ATTRIBUTES column in Section E.1. In addition, if the identifier is not defined as global in any of the compile units contained in the report, the ATTRIBUTES column contains an X for "external to program."

The DEFINED IN DECK column usually contains the compile unit name in which the identifier was declared; however, if the identifier was declared in an insert file, then the DEFINED IN DECK column contains the insert file name in which the identifier was declared. In this case the DEFINED IN DECK column also contains an I, indicating this column contains the name of an insert file. This column has the nine character restriction.

The USED IN DECK column contains a compile unit name in which the identifier (listed in the SYMBOL NAME column) was used. This column has the nine character restriction.

The USED IN PROCEDURE column contains the procedure name in which the indentifier (listed in the SYMBOL NAME column) was used. This column has the nine character restriction.

The REFERENCES USED column contains how many times the procedure used the identifier without changing it.

The REFERENCES CHNG column contains how many times the procedure changed the identifier. Changed identifiers are those that appear on the left-hand side of an assignment operator. They can also be changed by functions like PUSH and POP.

The following is an example Global Cross Reference Report.

```
GLOBAL CROSS REFERENCE The User's LEQ 36 Character Title      PAGE   1

SYMBOL    ATTRIBUTES    DEFINED       USED IN    USED IN    REFERENCES
 NAME                   IN DECK         DECK    PROCEDURE   USED  CHNG

ADD        P(0)         DEMOOPER
                                      DEMOMAIN   MAIN          1

ELSEIF    SYNONYM       EXTEND    I   DEMOMAIN   MAIN          4

ENDDO     SYNONYM       EXTEND    I   DEMOMAIN   MAIN          1

MAIN       P(0)         DEMOMAIN

OPEN      X                           DEMOMAIN   MAIN          2

PKB       PV            DEMOMAIN      DEMOMAIN   MAIN          1     1
                                      DEMOOPER   READINPUT     2

READ      X                           DEMOMAIN   MAIN          1
                                      DEMOOPER   READINPUT     2
```

F.2 *PROCEDURE CALL TREE REPORT*

The procedure call tree report presents an overview of a program's procedure calling structure. It shows which procedures call which procedures.

F.2.1 Producing a Procedure Call Tree Report

The notation .EXT; implies the highest version for input and the creation of a new version for output.

The procedure call tree report is developed from the tree data files that result when TRICOMP compiles a compile unit. These files by default have a .TRE; extension. This report is generated in a two-step process.

The first step is to combine the .TRE; files and the current master procedure call tree data file into a new master procedure call tree data file. The master procedure call tree data file by default has a .MTD; extension. Updating the master procedure call tree data file is done by invoking TREUPDATE. By default TREUPDATE executes under the assumption that all .TRE; files to be combined reside in the current default directory. The following command invokes TREUPDATE:

```
    $ TREUPDATE[/qualifier] file-spec
```

where file-spec is master procedure call tree data file. If the file-spec is just Filename, then TREUPDATE combines the .TRE; files and the current Filename.MTD; file into a new Filename.MTD; file.

TREUPDATE has the following command qualifier:

1. /TRE=(file-spec[,...]) - This requests that only the selected files be used to update the master procedure call tree data file.

The second step is to transform the master procedure call tree data file into a readable procedure call tree report. This is done by invoking TREREPORT with the following command.

```
$ TREREPORT[/qualifier...] file-spec
```

where file-spec is the master procedure call tree data file produced by the previous step. The file produced by TREREPORT is the master procedure call tree report, which by default has a .MTR; extension. This report can be examined and/or printed.

TREREPORT has the following two command qualifiers:

1. /TITLE="The User's LEQ 36 Character Title" - This requests that "The User's LEQ 36 Character Title" be placed on the top of each page of the report. The default title 'Procedure Call Tree Report".

2. /MTR=mtr-file-spec - This requests that the procedure call tree report be placed in mtr-file-spec. Default is Filename.MTR.

This two-step process allows incremental updates to the procedure call tree data. As the need arises to change selected compile units, the .THL files are changed, compiled using TRICOMP, assembled, and linked. Later, the procedure call tree data files are gathered by invoking TREUPDATE to do a partial update to the master procedure call tree data file. TREREPORT is used to form the new report. The changed compile units is reflected accordingly in the new report.

The master procedure call tree data file is a running record of the procedure call structure in a program. It is important not to delete this file. Each invocation of TREUPDATE and TREREPORT creates a new version of the master procedure call tree data and master procedure call tree report files respectively.

F.2.2 Reading a Procedure Call Tree Report

The procedure call tree report shows which procedures call which procedures. The report is arranged so that each page is divided into two major columns, each of which has three subcolumns. Each page has a header of the following form:

```
PROCEDURE CALL TREE    The User's LEQ 36 Character Title      PAGE   1
```

The columns and subcolumns are not marked with headers, but they are obvious as to where they are. Each of the two major columns are identical, so only the subcolumns are described. For the purpose of discussing them, the three subcolumns are assumed to have a header of the following form:

```
COLUMN1   COLUMN2   COLUMN3
```

Note that these columns contain at most 9 characters. If a procedure or compile unit name contains less than 10 characters, then the entire procedure or compile unit name appears in this column. If the procedure or compile unit name contains more than 9 characters, then the first 8 characters followed by an asterisk appear in that column.

COLUMN1 contains the procedure name. All procedures that are declared within a THLL program are listed alphabetically in this column.

COLUMN2 contains the following:

G	Definition line for a global procedure
blank	Definition line for a local (non-global) procedure
procedure name	The procedure called

COLUMN3 contains the following:

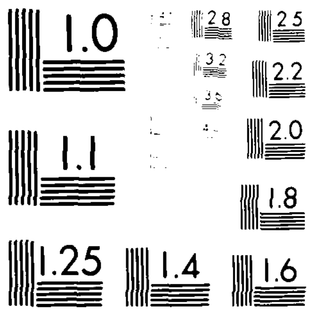
cu name	Compile Unit name in which the current procedure is defined
blank	Called procedure is global
L	Called procedure is local to the compile unit and not global
X	Called procedure is not defined within the program. It comes from a library or non-THLL object.
*	The called global or local procedure calls no procedures itself

The following is an example Procedure Call Tree Report.

AD-A145 116 THLL (TRIDENT HIGHER LEVEL LANGUAGE) REFERENCE MANUAL 3/3
(U) NAVAL SURFACE WEAPONS CENTER DAHLGREN VA H G HUBER
JUL 84 NSWC/TR-84-101

UNCLASSIFIED F/G 9/2 NL

END
DATE
FILMED
9-84
DTIC



MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS 1963 A

```
PROCEDURE CALL TREE The Users LEQ 36 Character Title        PAGE    1

 ADD        G          DEMOOPER
            READINPUT L
            WRITE      X

 DIVIDE     G          DEMOOPER
            READINPUT L
            WRITE      X

 MAIN       G          DEMOMAIN
            ADD
            DIVIDE
            MULTIPLY
            OPEN       X
            READ       X
            SUBTRACT
            WRITE      X

 MULTIPLY   G          DEMOOPER
            READINPUT L
            WRITE      X

 READINPUT             DEMOOPER
            WRITE      X
            READ       X

 SUBTRACT   G          DEMOOPER
            READINPUT L
            WRITE      X
```

F.3 NESTED PROCEDURE CALL TREE REPORT

The nested procedure call tree report presents an overview of a program's procedure calling structure. It shows which procedures call which procedures in a nested format.

F.3.1 Producing a Nested Procedure Call Tree Report

The notation .EXT; implies the highest version for input and the creation of a new version for output.

The nested procedure call tree report is developed from the tree data files that result when TRICOMP compiles a compile unit. These files by default have a .TRE; extension. This report is generated in a two-step process. The first step is to combine the .TRE; files into a Filename.MTD; master tree data file. This is done by using TREUPDATE as shown above.

The second step is to transform the master procedure call tree data file into a readable nested procedure call tree report. This is done by invoking NTREREPORT with the following command.

```
$ NTREREPORT[/qualifier...] file-spec
```

where file-spec is the master procedure call tree data file. The file produced by NTREREPORT is the nested procedure call tree report, which by default has a .NTR; extension. This report can be examined and/or printed.

NTREREPORT has the following command qualifiers:

1. /TITLE="The User's LEQ 36 Character Title" - This requests that "The User's LEQ 36 Character Title" be placed on the top of each page of the report. The default title is "NESTED PROCEDURE CALL TREE REPORT".

2. /NTR=ntr-file-spec - This requests that the nested procedure call tree report be placed in ntr-file-spec. Default is Filename.NTR.

3. /LASER - This requests a report that can be used for laser printing. This report has extra page ejects that make the laser listing more aesthetically appealing.

   /NOLASER - This qualifier requests a report that will not be used for laser printing. DEFAULT.

4. /XREF - This requests a list of cross reference tables, one for each procedure. Each table (for a procedure P) lists in one column all procedures that P calls and in another column all procedures that call P.

   /NOXREF - This requests that no cross reference tables be listed. DEFAULT.

5. /LONG - This requests a long tree. In this case the procedures called by any procedure P are listed for each occurrence of P in the tree.

   /NOLONG - This requests a short (or no long) tree. In a short tree, the procedures called by P are expanded only once. The next time P is listed the line number of its first expansion is listed. DEFAULT.

6. /OPT=opt-file-spec - This qualifier requests a tree as directed by the options contained in the opt-file-spec. The default file extension is .OPT. This file contains a sequence of option lines and data lines. An option line contains a / in column 1. A data line does not contain a / in column 1. The following option lines may be placed in the option file.

1. /LASER - This option line is the same as the /LASER command qualifier.

2. /XREF - This option line is the same as the /XREF command qualifier.

3. /LONG - This option line is the same as the /LONG command qualifier.

4. /ROOTS - This option line allows the user to declare a set of root procedures. A procedure call tree is produced for each declared root. The roots are declared on a sequence of data lines following the /ROOTS option line. The format of a data line is:

   ```
   procedure_name compile_unit_name
   ```

5. /LIBS - This option line allows the user to declare a set of library procedures. These procedures are not listed in the tree. If a library procedure is not a terminal node (normally it is) then the entire subtree of this library procedure is omitted. The library procedures are declared on a sequence of data lines following the /LIBS option line. The format of a data line is:

   ```
   procedure_name compile_unit_name
   ```

   If the procedure is external to the program, then the compile_unit_name must be left blank.

6. /TIPS - This option line allows the user to declare a set of terminal procedures. These procedures are listed in the tree as terminal nodes and marked with a "+" meaning a subtree has been pruned. The terminal procedures are declared on a sequence of data lines following the /TIPS option line. The format of a data line is:

   ```
   procedure_name compile_unit_name
   ```

   If the procedure is external to the program, then the compile_unit_name must be left blank.

NOTE: The procedure_name or compile_unit_name can be up to 9 characters. If one is longer than 9 characters, it should be listed on a data line as 9 characters. The first 8 characters listed must be the first 8 characters of the name and the 9th character must be an asterisk. For example, suppose a procedure LONGNAMEPROC of compile unit LONGNAMECU is to be on a data line. It should appear as the following:

```
LONGNAME* LONGNAME*
```

A procedure that does not occur as a called procedure is a "root" procedure. A report is generated for each root that is encountered in the master procedure call tree data file.

This two-step process allows incremental updates to the procedure call tree data. As the need arises to change selected compile units, the .THL files are changed, compiled using TRICOMP, assembled, and linked. Later, the procedure call tree data files are gathered by invoking TREUPDATE to do a partial update to the master procedure call tree data file. NTREREPORT is used to form the new nested procedure call tree report. The changed compile units are reflected accordingly in the new nested procedure call tree report.

The master procedure call tree data file is a running record of the procedure call structure in a program. It is important not to delete this file. Each invocation of TREUPDATE and NTREREPORT creates a new version of the master procedure call tree data and nested procedure call tree report files respectively.

F.3.2 Reading a Nested Procedure Call Tree Report

The nested procedure call tree report shows which procedures call which procedures in a nested format. Each page has a header of the following form:

```
PROCEDURE CALL TREE The User's LEQ 36 Character Title     PAGE   1
MAIN TREE                                       26-MAR-1984 12:26:19


LINE  LEVEL ROUTINE
----  ----- -------
```

The LINE column contains a running line number of the nested procedure call tree report.

The LEVEL column contains the level of the procedure which is defined as follows:

1. The root procedure is level 1.
2. If procedure A is level N and it calls procedure B, then procedure B is level N+1.

The ROUTINE column contains the procedure name. This column contains at most 9 characters. If a procedure or compile unit name contains less than 10 characters, then the entire procedure or compile unit name appears in this column. If the procedure or compile unit name contains more than 9 characters, then the first 8 characters followed by an asterisk appear in this column. The procedure name is indented (from the ROUTINE start column) by [LEVEL-1]*3 columns.

The procedure name contained in the ROUTINE column can be preceded by one of the following marks:

1. * - This indicates the procedure P is recursive. In this case the line number of the first occurence of P is enclosed in parentheses immediately following P.

2. - - This indicates the procedure P is external to the program.

3. + - This indicates the procedure P is a user declared terminal procedure (via the /TIPS option of the opt-file-spec).

The following is an example Nested Procedure Call Tree Report.

```
PROCEDURE CALL TREE The User's LEQ 36 Character Title     PAGE    1
MAIN TREE                                         26-MAR-1984 12:26:19


LINE  LEVEL ROUTINE
----- ----- -------

    1     1 MAIN
    2     2    ADD
    3     3       READINPUT
    4     4          -READ
    5     4          -WRITE
    6     3         -WRITE
    7     2    DIVIDE
    8     3       READINPUT
    9     4          -READ
   10     4          -WRITE
   11     3         -WRITE
```

If the /XREF command qualifier is included on the NTREREPORT command, a cross reference listing is also generated. Each page of the cross reference listing has a header of the following form:

```
PROCEDURE CALL TREE The User's LEQ 36 Character Title     PAGE    1
CROSS REFERENCE LISTING                           26-MAR-1984 12:26:19


P1             CALLS P2           AND IS CALLED BY P3
                     P4                            P5
```

The procedures in a program are arranged alphabetically. For each procedure P in the program, there are two columns (arranged alphabetically), the first listing the procedures called by P and the second listing the procedures that call P.

The following is an example Cross Reference Listing as produced by NTREREPORT.

```
PROCEDURE CALL TREE The User's LEQ 36 Character Title     PAGE    1
CROSS REFERENCE LISTING                        26-MAR-1984 12:26:19


ADD            CALLS READINPUT   AND IS CALLED BY MAIN
                     WRITE

MAIN           CALLS ADD         AND IS CALLED BY NONE
                     SUBTRACT

READ           CALLS NONE        AND IS CALLED BY READINPUT

READINPUT      CALLS READ        AND IS CALLED BY ADD
                     WRITE                        SUBTRACT

SUBTRACT       CALLS READINPUT   AND IS CALLED BY MAIN
                     WRITE

WRITE          CALLS NONE        AND IS CALLED BY ADD
                                                  READINPUT
                                                  SUBTRACT
```

APPENDIX G

THLL IN THE VAX/VMS ENVIRONMENT

G.1 INTRODUCTION

Filenames under VMS have the format NODE::DISK:[DIR]Basename.EXT;VER. Basename is the base filename for a set of related files that have different .EXT extensions. Most VAX programs have a set of conventions governing the .EXT extensions. This is also true for THLL programs. The user can override these conventions, but this is discouraged for configuration management reasons. The organization of a THLL program as presented in this appendix is not the only organization, but it is a logical one that should be considered seriously before deviating to another.

G.2 THE TRICOMP COMMAND

THLL in the VAX/VMS environment is implemented by the TRICOMP compiler, which is invoked by the TRICOMP command. The command TRICOMP translates a THLL compile unit into VAX MACRO, BP assembler or binary, or MC68000 assembler code. In addition to creating the object code, TRICOMP creates a listing, global cross reference data, and procedure call tree data. The command has the following form:

```
TRICOMP[/qualifier...] input-file-spec
```

where input-file-spec specifies the name of the file containing a THLL compile unit.

G.2.1 <u>Qualifiers</u>

TRICOMP command options are provided in the form of command qualifiers. Following is a list of qualifiers that can be used with the TRICOMP command.

```
/VAX
```

/VAX requests translation of THLL into VAX MACRO code. DEFAULT. Note that if /VAX is used, it should be the first qualifier. Also note that /VAX, /BP, and /MC are mutually exclusive.

/BP

/BP requests translation of THLL into BP assembly or binary code. Note that if /BP is used, it should be the first qualifier. Also note that /VAX, /BP, and /MC are mutually exclusive.

/MC

/MC requests translation of THLL into MC68000 assembler code. Note that if /MC is used, it should be the first qualifier. Also note that /VAX, /BP, and /MC are mutually exclusive.

/ALL or /NOALL

/ALL requests that all compile units (in a file) be compiled. Normally, only one compile unit should be contained in a file. The MACRO assembler assembles only the first compile unit in the resultant .MAR file. This qualifier is used mainly in compiler testing.

/NOALL requests that only one compile unit (in a file) be compiled. DEFAULT.

/BIN[=bin-file-spec] or /NOBIN

/BIN requests the BP binary data be placed in file bin-file-spec. The DEFAULT bin-file-spec is the file name of the input-file-spec with a .BIN extension. /BIN is valid only for TRICOMP/BP. DEFAULT.

/NOBIN suppresses the BP binary data file. /NOBIN is valid only for TRICOMP/BP.

/CGOPTS=oct-number

/CGOPTS is used to activate various compiler code generator debug dumps. The meaning of the various bits in the oct-number is compiler-dependent.

/DEBUG=oct-number

/DEBUG is used to activate various compiler debug dumps in pass 1 and pass 2. The meaning of the various bits in the oct-number is compiler-independent.

/GXR[=gxr-file-spec] or /NOGXR

/GXR requests the global cross reference data be placed in file gxr-file-spec. The DEFAULT gxr-file-spec is the file name of the input-file-spec with a .GXR extension. DEFAULT.

/NOGXR suppresses the global cross reference data file.

/MAR[=mar-file-spec] or /NOMAR

/MAR requests the MACRO assembly data be placed in file mar-file-spec. The DEFAULT mar-file-spec is the file name of the input-file-spec with a .MAR extension. /MAR is valid only for TRICOMP/VAX. DEFAULT.

/NOMAR suppresses the MACRO assembly data file. /NOMAR is valid only for TRICOMP/VAX.

/MCS[=mcs-file-spec] or /NOMCS

/MCS requests the MC68000 assembly data be placed in file mcs-file-spec. The DEFAULT mcs-file-spec is the file name of the input-file-spec with a .MCS extension. /MCS is valid only for TRICOMP/MC. DEFAULT.

/NOMCS suppresses the MC68000 assembly file. /NOMCS is valid only for TRICOMP/MC.

/SRC[=src-file-spec] or /NOSRC

/SRC requests the BP assembly data be placed in file src-file-spec. The DEFAULT src-file-spec is the file name of the input-file-spec with a .SRC extension. /SRC is valid only for TRICOMP/BP.

/NOSRC suppresses the BP assembly data file. /NOSRC is valid only for TRICOMP/BP.

/TLS[=tls-file-spec]

/TLS requests the TRICOMP listing be placed in file tls-file-spec. The DEFAULT tls-file-spec is the file name of the input-file-spec with a .TLS extension.

/TRE[=tre-file-spec] or /NOTRE

/TRE requests the procedure call tree data be placed in file tre-file-spec. The DEFAULT tre-file-spec is the file name of the input-file-spec with a .TRE extension. DEFAULT.

/NOTRE suppresses the procedure call tree data file.

G.2.2 Input To TRICOMP

G.2.2.1 Compile Units (.THL Files) - By default TRICOMP expects a compile unit in a file with a .THL extension. Each .THL file contains one THLL compile unit. A .THL file is an input source file that can be maintained with a VAX editor. It is recommended that each .THL file have a base filename corresponding to the compile unit name contained in the file.

G.2.2.2 Insert Files (.THI Files) - In addition to the .THL files that correspond to compile units, a THLL compile unit often includes INSERT files. An insert file is an input source file that must have a .THI extension. It can be created simply by editing a file. In order to locate a .THI file associated with the insert declaration:

```
INSERT FILENAME(DIRNAME)
```

TRICOMP searches for DIRNAME in the following order:

1. The VMS logical name table is searched for a definition of DIRNAME. If it exists, that name is the name of the directory that contains the FILENAME.THI file. The DEFINE command in VMS is used to establish this relationship in the logical name table.

2. If DIRNAME is the default identifier, DEFAULT, the current default directory contains the FILENAME.THI file. DEFAULT could be defined in the logical name table in which case the check above would have associated DEFAULT to a (possibly different) directory.

3. Finally, it is assumed that DIRNAME is a subdirectory in the current default directory.

This approach gives the program designer quite a bit of flexibility.

NOTE

DIRNAME and FILENAME are truncated to 8 characters.

G.2.3 Output From TRICOMP (.TLS, .GXR, .TRE, Object FILES)

The two file types, .THL and .THI, described above are created by the users as they develop their program. A .THL file contains a compile unit and is the input file to a single invocation of TRICOMP. The .THI files contain information to be INSERTed into .THL files during a THLL compilation. TRICOMP compiles a single .THL file containing a single compile unit and produces four files. The four files created by TRICOMP contain the following:

1. The compiled listing file. By default this file has a .TLS extension.

2. The global cross reference data file. By default this file has a .GXR extension.

3. The procedure call tree data file. By default this file has a .TRE extension.

4. The object file. This file extension varies according to which compiler was selected (VAX, BP, or MC68000). See the appropriate User's Guides (References 1, 2, and 3).

By default the base filename of each of these files is that of the .THL file containing the compile unit. TRICOMP creates the four output files in the current default directory by appending the above extensions to the filename of the .THL file. The TRICOMP command qualifiers provide wa to override the defaults.

G.3 ORGANIZATION OF A THLL PROGRAM

A program written in THLL consists of one or more compile units (along with the various insert files used by the compile units). A compile unit corresponds to a module, which is a set of related procedures and data that can be compiled separately. A single compile unit maps nicely into a single EDT file. Just as a compile unit maps into a file, a program (which is a collection of compile units) maps into a directory (which is a collection of files). Thus, a logical organization for a program written in THLL is one or more .THL files (each of which contains one compile unit) in a common directory.

In addition to placing a program's compile units in a common directory, insert files, listing files, global cross reference files, and procedure call tree files must be placed somewhere. If the insert files are used only by the one program, they can logically be placed in a subdirectory of the program's directory, whereas if the insert files are used by several programs, they can logically be placed in a parallel directory. For each file in the program's directory containing a compile unit, there is a listing file, an object file, a global cross reference file, and a procedure call tree file. By default, TRICOMP places these files in the current default directory. If a program consists of a manageable number of compile units, then all of the files can be left in the program's directory; otherwise the TRICOMP command qualifiers can be used to distribute them into different (possibly parallel) directories.

G.4 $SEVERITY RETURNED FROM TRICOMP

When TRICOMP exits to VMS, $SEVERITY is set to indicate the results of the compilation. The following values can be found in $SEVERITY.

1. $SEVERITY = 1 implies a normal compilation
2. $SEVERITY = 0 implies normal compilation with compile errors
3. $SEVERITY = 2 implies abnormal compilation
4. $SEVERITY = 4 implies one of the pertinent files could not be found or opened

Note that if the \\ ABORT directive is contained within a compile unit, then a $SEVERITY of 0 becomes a 2 and a $SEVERITY of 2 becomes a 4.

INDEX

DISTRIBUTION

Library of Congress Attn: Gift and Exchange Division Washington, DC 20540	(4)
General Electric Company Ordnance Systems 100 Plastics Avenue Pittsfield, MA 01201 Attn: G. Desmarais	(6)
J. Fenton	(6)
Strategic Systems Program Office Department of the Navy Washington, DC 20376 Attn: SP-23115 (C. Chappell)	(1)
EG&G Washington Analytical Services Center P.O. Box 552 Dahlgren VA 22448 Attn: IMC	(2)
Local:	
K50-GE	(1)
K51	(2)
K52	(12)
K53	(50)
K54	(20)
E31 (GIDEP Office)	(1)
E431	(10)